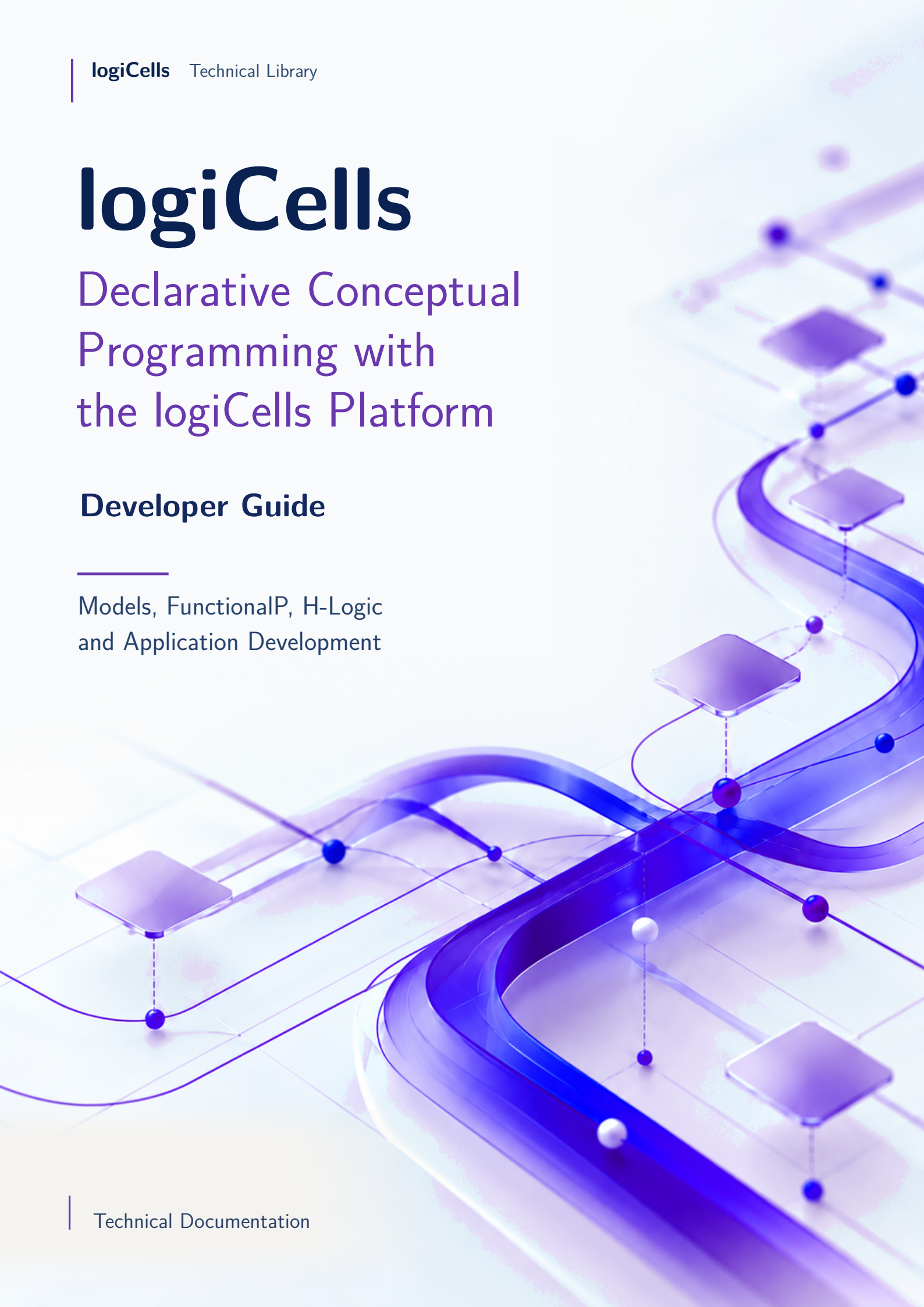
logiCells

Declarative Conceptual Programming with the logiCells Platform

Developer Guide

Models, FunctionalP, H-Logic and Application Development



Contents

About This Developer Guide **20**

I Start Here: The Executable Conceptual Model **21**

1 The logiCells Mental Model **22**
1.1 Applications as Conceptual Systems . . . 22
1.2 Concepts . . . 22
1.3 Facets . . . 23
1.4 Relations and Role Paths . . . 23
1.5 Three Kinds of Values: Scalar, Object Reference and Relation . . . 24
1.6 Actions . . . 24
1.7 Events . . . 25
1.8 Collections and Views . . . 25
1.9 The Runtime . . . 25

2 First Success: Install, Declare, Validate **26**
2.1 Check the tooling . . . 26
2.2 Create one complete MODEL source . . . 26
2.3 Validate and inspect . . . 27

3 Application Interfaces and Architectural Choices **28**
3.1 CLI . . . 28
3.2 API and services . . . 28
3.3 MVP or another custom interface . . . 28
3.4 logiCells Views . . . 29
3.5 External Host-Language Integration . . . 29
3.6 Decision rule . . . 29

4 Documentation, Compatibility and Scope **30**
4.1 Selecting an Authoritative Contract . . . 30
4.2 Reading Capability Descriptions . . . 30
4.3 Declarative Grammar and H-Logic . . . 30
4.4 Platform and Integration Boundaries . . . 31
4.5 Before Using an Example in Production . . . 31

II The Conceptual Programming Shift **32**

5 Why Declarative Conceptual Programming? **33**
5.1 The Fragmentation Problem . . . 33
5.2 Why Code-Centric Systems Lose Business Meaning . . . 34

5.3 From Data Models to Conceptual Models . . . 35
5.4 Declarative Does Not Mean Passive . . . 35
5.5 logiCells as an Executable Conceptual Platform . . . 35
5.6 When Declarative Conceptual Programming Is Useful . . . 35

6 Reading Map and Pattern Coherence 36
6.1 The Core Chain . . . 36
6.2 Platform Patterns and Application Patterns . . . 36
6.3 Functional Components and Appearance . . . 37
6.4 What Holds the Whole Architecture Together . . . 37

7 Conceptual Programming End-to-End Walkthrough 38
7.1 Why a Walkthrough Matters . . . 38
7.2 The Business Scenario . . . 38
7.3 Declaring the Scheduled Item . . . 38
7.4 Adding the Persistence Projection . . . 39
7.5 Declaring the Object-Level Action . . . 39
7.6 Declaring the Scheduler Manager . . . 40
7.7 What This Scenario Demonstrates . . . 40
7.8 Where to Go Next . . . 41

8 From Code-Centric Systems to Concept-Centric Systems 42
8.1 The Code-Centric Reflex . . . 42
8.2 Where Business Meaning Goes . . . 42
8.3 From Objects to Concepts . . . 42
8.4 From Tables to Conceptual Memory . . . 43
8.5 From Routes to Capabilities . . . 43
8.6 From Screens to Views . . . 43
8.7 Consequences for Architecture . . . 43

9 Applications as Executable Conceptual Models 44
9.1 From Application as Program to Application as Model . . . 44
9.2 What Makes a Model Executable . . . 44
9.3 The Application Boundary . . . 45
9.4 Actions as Part of the Model . . . 45
9.5 Views as Optional Projections . . . 45
9.6 Why Executability Matters . . . 45

10 Concepts before Classes, Tables and Routes 46
10.1 The Technical Reflex . . . 46
10.2 The Conceptual Order . . . 46
10.3 Concepts before Classes . . . 46
10.4 Concepts before Tables . . . 47
10.5 Concepts before Routes . . . 47
10.6 Concepts before Screens . . . 47
10.7 Why This Discipline Matters . . . 47

11 Metadata, Models and Runtime Meaning 48
11.1 Metadata Is Not Decoration . . . 48
11.2 Declarative Structure . . . 48
11.3 Three Authorities for Three Questions . . . 49
11.4 From Descriptive Metadata to Executable Metadata . . . 49

11.5 Model Meaning and Runtime Meaning . . . 49
11.6 Conceptual Metadata versus Technical Metadata . . . 49
11.7 YAML MODEL Source and the Declarative Contract . . . 50
11.8 A Model Is Not a Diagram . . . 50

12 The Executable Declarative Grammar 51
12.1 Supported declarative file families . . . 51
12.2 How named YAML items are encoded . . . 51
12.3 MODEL root contract . . . 52
12.4 Canonical actions . . . 52
12.5 Events are not action declarations . . . 52
12.6 Collections and role paths . . . 52
12.7 Compatibility versus canonical authoring . . . 53
12.8 Public names versus implementation bindings . . . 53

13 Migrating Existing XML Metadata to YAML 54
13.1 The Migration Pipeline . . . 54
13.2 Compare Meaning, Not Formatting . . . 54
13.3 Known Migration Hazards . . . 55
13.3.1 Hash-prefixed values . . . 55
13.3.2 Compact versus expanded items . . . 55
13.3.3 Polymorphic sequences . . . 55
13.3.4 Historical wrappers . . . 55
13.3.5 Embedded collection data . . . 55
13.4 Migrating Legacy Views to Interaction Contracts . . . 55
13.5 Three Levels of Validation . . . 56
13.6 What Not to Migrate Automatically . . . 56
13.7 Round-trip Regression Tests . . . 56

III Conceptual Modeling Foundations 57

14 Concepts, Instances and Relations 58
14.1 Concepts . . . 58
14.2 Instances . . . 59
14.3 Conceptual Identifiers . . . 59
14.4 Facets . . . 59
14.5 Relations . . . 59
14.6 Constraints . . . 60
14.7 Concepts, Instances and Relations in Execution . . . 60

15 Types, Roles and Polyadic Relations 61
15.1 From Fields to Conceptual Roles . . . 61
15.2 Role Names Are Part of the Model . . . 62
15.3 Binary Relations . . . 62
15.4 Polyadic Relations . . . 63
15.5 Relation Identity . . . 64
15.6 Polyadic Maps . . . 64
15.6.1 Explicit Maps, Derived Maps and Non-applicable Constraints . . . 65
15.7 Design Rules . . . 65

16 Identity, Context and Conceptual Scope 66
16.1 Conceptual Identity . . . 66
16.2 Instance Identity . . . 66
16.3 Relation Identity and this . . . 67
16.4 Entity Identity and me . . . 67
16.5 Context . . . 68
16.6 Conceptual Scope . . . 68
16.7 Redefinition and Extension . . . 68

17 Constraints, Defaults and Validity 69
17.1 Concept Constraints . . . 69
17.2 Participant Typing and Role Paths . . . 69
17.3 Defaults . . . 70
17.4 Nullability . . . 70
17.5 Action Parameter Contracts . . . 71
17.6 Fixed Business Invariants . . . 71
17.7 Validity Is Layered . . . 71

18 Conceptual Graphs and Hypergraph Memory 73
18.1 Why Hypergraph Memory Matters . . . 73
18.2 Concepts, Relations and Hyperedges . . . 74
18.3 A YAML Example: Hypergraph and Database Facets . . . 74
18.4 Hypergraph Memory and Runtime Execution . . . 75
18.5 Relation with AI and Semantic Memory . . . 75
18.6 Persistent Structural Pattern Evaluation . . . 75

19 Conceptual Layers, Projections and Views 76
19.1 The Concept Is Not the Projection . . . 76
19.2 Facets as Projection Boundaries . . . 77
19.3 Role Paths as Cross-Layer Bridges . . . 77
19.4 Interaction Contracts as Model Projections . . . 78
19.5 Layered Evolution . . . 78

IV H-Logic and Declarative Expressions 79

20 H-Logic Essentials 80
20.1 Why H-Logic Matters . . . 80
20.2 Relationship with YAML . . . 81
20.3 Terms, Types and Structural Use . . . 81
20.4 Roles and Attributes . . . 81
20.5 Applying a Known Type with Local Role Names . . . 82
20.6 Interface Attachment and Boxing . . . 82
20.7 Named Terms . . . 82
20.8 Quantifiers . . . 83
20.9 Logical Relations and Complements . . . 83
20.10Commands and Queries . . . 83
20.11Composite and Projected Definitions . . . 84
20.12Where H-Logic Adds the Most Value . . . 84
20.13H-Logic as a Canonical Exchange Form . . . 84

21 Declaring Concepts and Instances 86
21.1 The Class Declaration . . . 86
21.2 Concepts inside a Class . . . 86
21.3 Entity Class Type and Entity Concept . . . 87
21.4 Implicit Me Identity for Entities . . . 87
21.5 Implication Is a Semantic Mapping . . . 88
21.6 Conceptual Fields . . . 88
21.7 Instances as Runtime Occurrences . . . 88
21.8 Database Facets and Instances . . . 89
21.9 H-Logic Reading . . . 89

22 Declaring Relations and Reified Facts 90
22.1 Role References . . . 90
22.2 Reified Relations . . . 91
22.3 Why Reify a Relation? . . . 91
22.4 Persistence of Reified Relations . . . 91
22.5 H-Logic Reading . . . 92
22.6 Role Paths as Conceptual Bridges . . . 92

23 Contexts, Interfaces and Named Terms 93
23.1 Contexts Are Ordinary Roles . . . 93
23.2 Role Order . . . 93
23.3 Interfaces Preserve Support Identity . . . 93
23.4 Direct Instance versus Interface Attachment . . . 94
23.5 Explicit Named Terms . . . 94
23.6 Fixed and Default Values . . . 94
23.7 Complex Role Domains . . . 94
23.8 Why These Distinctions Matter . . . 95

24 Queries over Conceptual Structures 96
24.1 Ask and Question-Mark Forms . . . 96
24.2 Typed Quantification . . . 96
24.3 Binding to a Named Term . . . 96
24.4 Logical Composition . . . 97
24.5 Query as an Executable Bridge . . . 97
24.6 Why H-Logic Queries Are Valuable . . . 97

25 H-Logic Query Modes, Scope and Result Semantics 98
25.1 The Three Query Modes . . . 98
25.2 Homomorphic Pattern Matching . . . 98
25.3 Unifying Mode . . . 99
25.4 Resolving Mode . . . 99
25.5 Scope Options . . . 99
25.6 Direction in the Type Lattice . . . 100
25.7 Inherited Instances and Mapped Views . . . 100
25.8 Polyadic Maps Preserve Equality Information . . . 101
25.9 Result Contract . . . 101
25.10 Persistent Iteration and Performance . . . 102
25.11 Practical Selection Guide . . . 102
25.12 H-Logic as the Single Knowledge Matching Authority . . . 102

26 Rules, Constraints and Tested Patterns 103
26.1 Constraints as Model Commitments . . . 103
26.2 Rules as Executable Interpretation . . . 103
26.3 Expression Constraints and Role Compatibility . . . 103
26.4 Tested Patterns . . . 104
26.5 Constraint Placement . . . 104
26.6 Design Rules . . . 104
26.7 Summary . . . 104

27 Common H-Logic Misreadings 105
27.1 A Role Name Is Not a Named-Term Reference . . . 105
27.2 Type Restriction and Equality Binding Are Separate . . . 105
27.3 Contexts Are Roles, Not Function Calls . . . 105
27.4 Colon and Double Colon Are Different . . . 106
27.5 Structural Uses Require Defined Types . . . 106
27.6 Query Variables Are Not Prefixed with Question Marks . . . 106
27.7 Role Order Is Declaration Metadata . . . 106
27.8 Composite Definitions Must Constrain Public Roles . . . 106
27.9 A Wildcard Is Not a Nil Constraint . . . 106
27.10Subtype Search Does Not Return Raw Descendants Unchanged . . . 107
27.11A Polyadic Map Is Not Only a Permutation . . . 107
27.12Local Role Names Do Not Redefine a Known Type . . . 107
27.13A Literal Is Not a Public Role . . . 107
27.14No Shared Role Can Mean No Valid Mapping . . . 107
27.15Nested Logical Groups Must Remain Groups . . . 107
27.16One Result Is Not Necessarily a One-Element Container . . . 107
27.17H-Logic Does Not Replace YAML, VIEW XML or functionalP . . . 108

V From Business Specifications to Applications 109

28 Two Paths from Business Specifications to logiCells Applications 110
28.1 Two Entry Points, Two Feedback Loops . . . 111
28.2 Path A: Model-First, H-Logic as a Semantic Synthesis . . . 111
28.2.1 Typical contexts for the model-first path . . . 112
28.3 Path B: H-Logic-First, Models as Operationalization . . . 112
28.3.1 Verbs and operationalization . . . 112
28.3.2 Typical contexts for the H-Logic-first path . . . 113
28.4 How the Two Paths Converge . . . 113
28.5 Choosing a Path . . . 114
28.6 Switching Loops Is Normal . . . 114
28.7 Keep Expertise Proportional to the Project . . . 115
28.8 Using Skills to Enter Either Loop . . . 115
28.9 Where to Continue . . . 116

29 From Business Specifications to Validated H-Logic 117
29.1 Why Introduce a Logical Pivot? . . . 117
29.2 The Three Representations . . . 118
29.2.1 Natural-language specification . . . 118
29.2.2 H-Logic semantic specification . . . 118
29.2.3 logiCells declarative artifacts . . . 118
29.3 The Three Validation Loops . . . 118

29.3.1 Human semantic review . . . 119
29.3.2 LLM-assisted inference under symbolic rules . . . 119
29.3.3 Syntax and semantic verification by the LGC engine . . . 119
29.4 Status and Provenance of Analysis Statements . . . 120
29.5 A Repeatable Construction Method . . . 120
29.5.1 Step 1: Write a compact specification . . . 120
29.5.2 Step 2: Extract and normalize the vocabulary . . . 120
29.5.3 Step 3: Decide where identity is required . . . 120
29.5.4 Step 4: Declare foundational types before structural use . . . 121
29.5.5 Step 5: Identify complete business vocabularies . . . 121
29.5.6 Step 6: Propagate time and context through the whole rule . . . 121
29.5.7 Step 7: Express calculations through typed support relations . . . 122
29.5.8 Step 8: Derive positive diagnostic facts . . . 122
29.5.9 Step 9: Design query access from real usage . . . 122
29.5.10 Step 10: Define expected deductions before generation . . . 123
29.6 The Role of Questions in LLM-Assisted Analysis . . . 123
29.7 Validation Gates . . . 123

30 Case Study: Temporal ERP Stock Analysis in H-Logic 124
30.1 Business Specification . . . 124
30.2 Review the Vocabulary Before Writing Rules . . . 124
30.3 Foundational Types and Complete Vocabulary . . . 125
30.4 Identified Temporal Business Entities . . . 125
30.5 Derived Facts and Support Relations . . . 126
30.6 Movement Classification . . . 126
30.7 Stock Consistency and Reorder Propagation . . . 127
30.8 Invalid States as Explainable Facts . . . 127
30.9 Derive Indexes from Query Patterns . . . 128
30.10 Manual Propagation Scenario . . . 129
30.11 Human Review Checklist . . . 129
30.12 LLM-Assisted Analysis Tasks . . . 130
30.13 Engine Test Plan . . . 130

31 From Validated H-Logic to Declarative Models 131
31.1 Operationalization, Not Textual Translation . . . 131
31.2 Generate Concepts from Identified Types . . . 131
31.3 Generate a Temporal Entity Model . . . 132
31.4 Generate Relations from Derived Facts . . . 133
31.5 Generation Decisions That H-Logic Does Not Make Alone . . . 133
31.6 Two Generation Modes . . . 134
31.6.1 Generation driven by logiCells . . . 134
31.6.2 Generation driven by LLM and skills . . . 134
31.7 Verification of Generated Artifacts . . . 135
31.8 End-to-End Traceability . . . 135
31.9 A Four-Level Test Strategy . . . 136
31.10 Recommended Project Artifacts . . . 136
31.11 Why This Workflow Matters . . . 136

VI Building Applications 137

32 Source Formats and Project Conventions 138
32.1 The source rule . . . 138
32.2 YAML field convention . . . 139
32.3 VIEW is deliberately different . . . 139
32.4 Migration rule for old forms and frames . . . 139
32.5 HTTP Page-Extension Resources . . . 139
32.6 Do not mix authorities . . . 140

33 Class Manifests: Loading, Modes and Conventions 141
33.1 What a Manifest Guarantees . . . 141
33.2 Manifest Attributes . . . 141
33.3 Loading a Class from a Multi-Class .models Container . . . 142
33.4 Choosing a Class Identifier Prefix . . . 142
33.5 Duplicates and Cross-Folder References . . . 143
33.6 Global Class Identifier Uniqueness . . . 143
33.7 Events in Manifests . . . 143
33.8 Recommended Validation . . . 144

34 Application Models 145
34.1 The Role of an Application Model . . . 145
34.2 Namespaces . . . 145
34.3 Design Rules . . . 145
34.4 Summary . . . 145

35 Collections and Conceptual Views 146
35.1 Collections as Navigable Meaning . . . 146
35.2 Views as Projections . . . 146
35.3 Collections, Cursors and Context . . . 146
35.4 Design Rules . . . 146
35.5 Summary . . . 146

36 Actions as Business Intentions 147
36.1 Why Actions Belong to Concepts . . . 147
36.2 Object-Level and Class-Level Actions . . . 147
36.3 Scripted and Native Implementations . . . 148
36.4 Parameters as Conceptual Boundaries . . . 148
36.5 Published Actions . . . 148
36.6 Actions in Processes . . . 148
36.7 Actions and Events . . . 148

37 functionalP for logiCells Beginners 150
37.1 Where functionalP Appears . . . 150
37.2 A Minimal Metadata Script . . . 150
37.3 Variables and Assignment . . . 151
37.4 Self, Sender and Result . . . 151
37.5 Accessing the PageControler . . . 151
37.6 Reading a Dataset . . . 151
37.7 Initializing Local Page State . . . 152
37.8 Filtering Before a Dataset Opens . . . 152
37.9 Posting the Current Object . . . 152

37.10Creating an Object . . . 152
37.11Dispatching a Visual Event . . . 153
37.12Calling a Visual Component Object-Behind . . . 153
37.13When to Move Logic Elsewhere . . . 153
37.14Beginner Mistakes . . . 153
37.15Practical Reading Path . . . 154

38 Design Decision Guide: What to Create and When 155
38.1 The Central Decision Chain . . . 155
38.2 Choose the Construction Path Before the Artifacts . . . 155
38.3 Decision Table . . . 156
38.4 Quick Questions Before Creating Anything . . . 157
38.5 Common Beginner Decisions . . . 157
38.5.1 Concept or Field? . . . 157
38.5.2 Relation or Foreign Key? . . . 157
38.5.3 Action or Event? . . . 157
38.5.4 Action or Workflow? . . . 158
38.5.5 View or Reactive Frame? . . . 158
38.6 Bad Pattern, Correction Pattern . . . 158
38.7 Where to Continue . . . 158

39 Events as Observable Conceptual Changes 159
39.1 Events and Actions . . . 159
39.2 Local and Remote Events . . . 159
39.3 Events as Process Boundaries . . . 160
39.4 Controllers as Event Observers . . . 160
39.5 Visual Events . . . 161

40 Workflows and Processes 162
40.1 Process as a Conceptual Object . . . 162
40.2 Synchronous and Asynchronous Coordination . . . 162
40.3 The Role of `lgc#process` . . . 163
40.4 Process Models as Anti-Complexity Boundaries . . . 163
40.5 Synchronous Process Example . . . 163
40.6 Asynchronous Process Example . . . 164
40.7 Design Rules . . . 165

41 Asynchronous Messaging as a Platform Pattern 166
41.1 Platform Messaging Concepts . . . 166
41.2 Message as a Conceptual Object . . . 166
41.3 Message States . . . 167
41.4 Event Dispatch from a Message . . . 167
41.5 Conversation as Coordination Context . . . 168
41.6 Local and Remote Messaging . . . 168
41.7 Packaging Pattern . . . 169

42 Roles, Permissions and Contextual Visibility 170
42.1 Roles as Conceptual Positions . . . 170
42.2 Permission Boundaries . . . 170
42.3 Contextual Visibility . . . 170
42.4 Design Rules . . . 170
42.5 Summary . . . 170

43 Application Modeling Design Rules 171
43.1 Separate Responsibilities . . . 171
43.2 Model Before Code . . . 171
43.3 Publish Intentionally . . . 171
43.4 Pair Examples . . . 171
43.5 Summary . . . 171

44 Runtime Objects, Containers, and Execution Contexts 172
44.1 From Declaration to Runtime Object . . . 172
44.2 Class Items and Object Items . . . 172
44.3 The Objects Container . . . 173
44.4 Self, Context, and Runtime Binding . . . 173
44.5 Persisting Conceptual Changes . . . 174
44.6 Design Rule . . . 174

45 Collections, Joins, and Queryable Models 175
45.1 Collections as Runtime Navigation Contracts . . . 175
45.2 Relations as Queryable Structures . . . 175
45.3 Join Collections . . . 176
45.4 Role Paths . . . 177
45.5 Queryable Models and Complexity . . . 177

46 Application Configuration and Package Assembly 178
46.1 Runtime Launch Configuration: `.lgc` YAML . . . 178
46.2 Application Identity and Desktop . . . 178
46.3 Graph and Metadata Loading . . . 179
46.4 Package Items, Schema Paths, View Paths and Desktops . . . 179
46.5 Database Aliases . . . 180
46.6 Binary Packages and Model Packages . . . 180
46.7 Platform and Application Namespaces . . . 180
46.8 Services and Files . . . 181
46.9 Reusable Platform Capabilities . . . 181
46.10 Local and Remote Runtime Resources . . . 182
46.11 From Package to Loaded Model . . . 182
46.12 Compiling and Starting an Application from the Command Line . . . 183
46.12.1 What Success Looks Like . . . 184
46.13 Design Rule . . . 184
46.14 Memory and Persistent Starter Configurations . . . 184
46.15 Compiled Metadata Versus Semantic Graph Initialization . . . 185

47 Incremental Compilation and Impact 186
47.1 Understand the Authority Before Selecting a Command . . . 186
47.2 Practical Developer Workflow . . . 186
47.3 What Changed: Source, Aspect and Dependency . . . 187
47.4 NativeStore Does Not Replace the Semantic Graph . . . 187
47.5 Cold, Warm and Physical YAML Cache . . . 187
47.6 Inspecting a Failure Without Weakening Invariants . . . 187
47.7 Impact Analysis in VS Code: The Correct Boundary . . . 188
47.8 Developer Verification Checklist . . . 188

48 Declarative Runtime Services and Engine Patterns 189
48.1 From Declaration to Runtime Behavior . . . 189
48.2 Three Service Levels . . . 189
48.3 Common Definition Attributes . . . 189
48.4 Additional Sections . . . 190
48.5 Built-In Architectural Patterns . . . 190
48.6 Example: HTTP Service Definition . . . 191
48.7 Virtual HTTP Pages and Page-Extension Files . . . 191
48.8 Adding a New Runtime Pattern . . . 191
48.9 Reference Strategy . . . 192

49 Deployment, Packaging, and Runtime Distribution 193
49.1 Supported Runtime Platforms . . . 193
49.2 Platform Availability versus Subsystem Certification . . . 193
49.3 The Deployment Tree . . . 194
49.4 Packaging Platform and Application Assets . . . 194
49.5 Configuration by Environment . . . 195
49.6 Updates and Runtime Evolution . . . 195
49.7 Containerized Deployment . . . 195
49.8 Design Rule . . . 196

50 Blueprint for a New ERP Module 197
50.1 1. Start from a bounded business capability . . . 197
50.2 2. Recommended module tree . . . 197
50.3 3. Model identity and invariants first . . . 197
50.4 4. Actions express reusable business intentions . . . 198
50.5 5. Collections are reusable read models . . . 198
50.6 6. Processes coordinate objects . . . 198
50.7 7. Add the VIEW package after the business contract . . . 198
50.8 8. Test the contract in layers . . . 199
50.9 9. Definition of done . . . 199

VII Runtime Semantics 200

51 From Declarations to Runtime Execution 201
51.1 The Runtime Path . . . 201
51.2 Class Items and Object Items . . . 201
51.3 Runtime Containers . . . 202
51.4 Facets at Runtime . . . 202
51.5 Actions as Runtime Entry Points . . . 203
51.6 Events and Reactive Execution . . . 203

52 Conceptual Memory and Change Propagation 204
52.1 Memory as a Conceptual Graph . . . 204
52.2 From Local Change to Propagation . . . 204
52.3 Observable Changes . . . 205
52.4 Design Rules . . . 205
52.5 Summary . . . 205

53 Evaluation Contexts **206**
53.1 Why Context Matters . . . 206
53.2 Common Contexts . . . 206
53.3 Context Propagation . . . 206
53.4 Design Rules . . . 206
53.5 Summary . . . 207

54 Transactions, Consistency and Identity **208**
54.1 Conceptual Identity . . . 208
54.2 Consistency Across Facets . . . 208
54.3 Transaction Boundaries . . . 208
54.4 Process-Level Consistency . . . 208
54.5 Design Rules . . . 209
54.6 When Metadata, Semantic Graph and SQL Change Together . . . 209
54.7 Summary . . . 209

55 Reactive Execution and Event Routing **210**
55.1 Events as Declarative Links . . . 210
55.2 Business Events . . . 211
55.3 Visual Events . . . 211
55.4 Process Events . . . 211
55.5 Messaging Events . . . 211
55.6 Event Sources and Inheritance . . . 211
55.7 Dispatching and Propagation . . . 212

56 Services, Actions and Publication **213**
56.1 Action as a Service Boundary . . . 213
56.2 The Declarative Signature . . . 214
56.3 Web Service Access . . . 214
56.4 Synchronous Service Calls . . . 214
56.5 Asynchronous Service Bridges . . . 214
56.6 Security and Publication Scope . . . 215
56.7 Design Rules . . . 215

57 Runtime Boundaries and Extension Points **216**
57.1 Model Boundary . . . 216
57.2 Implementation Boundary . . . 216
57.3 Publication Boundary . . . 216
57.4 Messaging Boundary . . . 216
57.5 Design Rules . . . 216
57.6 Implementation Bindings Are Not Business Concepts . . . 217
57.7 Summary . . . 217

58 Runtime Lifetime, Ownership and Shutdown **218**
58.1 Memory Lifetime and Operational Lifetime Are Different . . . 218
58.2 Ownership Families . . . 218
58.3 Service and Worker Shutdown Order . . . 218
58.4 ABI Ownership . . . 219
58.5 Resident AI Lifetime . . . 219
58.6 Rich Client Lifetime . . . 219
58.7 Developer Checklist . . . 219

VIII CLI, APIs and External Integration 221

59 Using the cell Command-Line Tool 222
59.1 Architecture . . . 222
59.2 Global options . . . 222
59.3 Start with doctor . . . 222
59.4 Workspace, application and package inspection . . . 222
59.5 MODEL authoring and inspection . . . 223
59.6 Semantic exploration . . . 223
59.7 Scripts, services, runtime and debugger . . . 223
59.8 Application publication: MCP and OpenAPI . . . 224
59.9 Agent execution and persistent sessions . . . 224
59.10 CLI v0.17: Incremental Diagnostics, Transformers and Documentation . . . 224
59.11 Do Not Conflate Agent and Developer-Agent Namespaces . . . 225
59.12 Contract rule . . . 225

60 API and Service Publication 226
60.1 Purpose . . . 226
60.2 From Action Signature to API Operation . . . 226
60.3 Request and Response Mapping . . . 226
60.4 Service Groups . . . 227
60.5 Published Actions over Web Services . . . 227
60.6 HTTP Virtual Pages and Route Selection . . . 227
60.7 Process-Oriented APIs . . . 227
60.8 Machine-Readable Contracts: OpenAPI and MCP . . . 227
60.9 Error and Validation Semantics . . . 228
60.10 Design Rules . . . 228

61 Publishing Applications through OpenAPI and MCP 229
61.1 Publication Starts with the Application Contract . . . 229
61.2 Compiling an Application for MCP . . . 229
61.2.1 Hosting the generated MCP surface . . . 230
61.3 Compiling an Application for OpenAPI . . . 230
61.4 Application and Package Publication . . . 230
61.5 The Two MCP Directions . . . 231
61.6 Type and Validation Discipline . . . 231
61.7 Recommended Developer Workflow . . . 231
61.8 Protocol Projection Is Not Authorization . . . 232
61.9 Inbound MCP Calls and Runtime Authority . . . 232
61.10 Reference Configuration Requirement . . . 232

62 Declarative External API Adapters 233
62.1 External APIs as Conceptual Boundaries . . . 233
62.2 Request, Response and Service Concepts . . . 233
62.3 A Generic Paginated API Pattern . . . 233
62.4 From API Adapter to Collection Provider . . . 234
62.5 Errors, Status and Traceability . . . 235
62.6 Design Rules . . . 235

63 Integration with External Systems 236
63.1 Integration as a Boundary Problem . . . 236
63.2 Synchronous Integration . . . 236
63.3 Asynchronous Integration . . . 236
63.4 Database Integration . . . 237
63.5 Message and Event Integration . . . 237
63.6 Agents and External Execution . . . 237
63.7 Design Rules . . . 237

64 Declarative Models and the Runtime ABI 238
64.1 Current Availability . . . 238
64.2 Stable Invocation Identity . . . 238
64.3 Value Model . . . 238
64.4 Ownership Is Part of the ABI Contract . . . 239
64.5 Complete Ownership Vocabulary . . . 239
64.6 Release Policies . . . 239
64.7 Python Binding . . . 240
64.8 C# Binding . . . 240
64.9 Parameters and Write-Back . . . 240
64.10Native and RPC Placement . . . 240
64.11ABI Manifests and Target Information . . . 241
64.12Platform Scope of Native ABI Loading . . . 241
64.13Validated ABI Behavior . . . 241
64.14Developer Rule . . . 241

65 Publication Design Rules 242
65.1 Publish Capabilities, Not Internals . . . 242
65.2 Use Typed Signatures . . . 242
65.3 Separate Publication Channels . . . 242
65.4 Versioning . . . 242
65.5 Summary . . . 242

IX logiCells and AI 243

66 Conceptual Memory for AI Applications 244
66.1 Purpose . . . 244
66.2 Beyond Document Retrieval . . . 244
66.3 Structured Context for AI . . . 244
66.4 Actions as Safe Execution Handles . . . 245
66.5 Conceptual Memory and Explanations . . . 245
66.6 Design Rules . . . 245

67 Positioning logiCells in the Neuro-Symbolic Design Space 246
67.1 The Kautz Taxonomy as an Architectural Map . . . 247
67.1.1 Symbolic–Neural–Symbolic Pipelines . . . 247
67.1.2 Neural Perception Followed by Symbolic Reasoning . . . 247
67.1.3 Symbolic Systems Generating Neural Supervision . . . 247
67.1.4 Symbolic[Neuro] . . . 248
67.2 logiCells's First Position: Governed Composition . . . 248
67.3 logiCells's Second Position: Symbolic[Neuro] Deduction . . . 250
67.4 Why the Hypergraph Changes the Integration Problem . . . 250

67.5 Hypergraphs, Not Merely Binary Graphs . . . 251
67.6 Design Commitments . . . 251
67.6.1 Neural Performance Must Not Be Confused with Formal Correctness . . . 252
67.6.2 Every Intuition Is Versioned . . . 252
67.6.3 Proposals Are Not Conclusions . . . 252
67.6.4 The Native Runtime Remains Focused . . . 252
67.7 Architectural Scope and Explicit Non-Goals . . . 253
67.8 Terminology and Primary References . . . 253
67.9 Current Status and Intended Evolution . . . 254
67.10 Positioning Summary . . . 254

68 Agents Operating on Conceptual Models 255
68.1 Purpose . . . 255
68.2 Core Ideas . . . 255
68.3 Publication Boundary and MCP . . . 256
68.4 Using VIEW Interaction Contracts without Rendering a Form . . . 256
68.5 Agent Invocation of Published Actions . . . 257
68.6 Agent and Process Coordination . . . 257
68.7 Agent and Messaging . . . 257
68.8 Human-in-the-Loop Automation . . . 257
68.9 AI Agents and the Symbolic Gate . . . 257
68.10 Design Rules . . . 257

69 Agent Runtime, Goals and Persistent Sessions 258
69.1 One-Shot Agent Execution . . . 258
69.2 Knowledge Inputs and H-Logic Queries . . . 259
69.3 Canonical Agent Runtime Terms . . . 259
69.4 Persistent Agent Sessions . . . 259
69.5 Goals . . . 260
69.6 Pending Confirmations . . . 260
69.7 Snapshots and Evaluation . . . 261
69.8 MCP in Agent Execution . . . 261
69.9 Local Providers . . . 261
69.10 Testing an Agent Scenario . . . 261
69.11 Typed External Effects and Their Evidence . . . 262

70 Governed Agent Execution Contract 263
70.1 Authority Model . . . 263
70.2 Protocol-Neutral Execution Context . . . 263
70.3 Governed Action Metadata . . . 264
70.4 PendingConfirmation Semantics . . . 264
70.4.1 Reject . . . 264
70.4.2 Confirm . . . 265
70.5 Outbound MCP Tool Execution . . . 265
70.6 Inbound MCP Published-Action Execution . . . 265
70.7 OpenAPI and Other Publication Adapters . . . 266
70.8 Required Regression Scenarios . . . 266
70.9 Developer Quickstart Acceptance Criterion . . . 266
70.10 Deployment Checks for Published Actions . . . 267
70.11 Developer Assistance and Knowledge Boundaries . . . 267

71 Human-in-the-Loop Conceptual Workflows 268
71.1 Human Tasks as Process Steps . . . 268
71.2 Views as Task Interfaces . . . 268
71.3 AI Assistance . . . 268
71.4 Design Rules . . . 268
71.5 Summary . . . 268

72 Explainability, Traceability and Governance 269
72.1 Explainability from Structure . . . 269
72.2 Traceability from Events and Logs . . . 269
72.3 Governance of Published Capabilities . . . 269
72.4 AI Governance . . . 269
72.5 Design Rules . . . 269
72.6 Summary . . . 270

X Conceptual Engineering Practices 271

73 Testing Conceptual Models 272
73.1 Purpose . . . 272
73.2 What to Test . . . 272
73.3 Structure a Declarative Test Package . . . 273
73.4 Declare a Test Suite as a Model . . . 273
73.5 One Test Case per Declared Action . . . 274
73.6 Assertions in functionalP . . . 274
73.7 Organize Suites by Contract . . . 275
73.8 Run Declarative Tests with sclgc . . . 275
73.9 Testing Published Actions . . . 276
73.10Testing Processes . . . 276
73.11Testing Asynchronous Behavior . . . 276
73.12Testing Views . . . 276
73.13Regression and Model Evolution . . . 276
73.14Summary . . . 277

74 Debugging Declarative Applications 278
74.1 Debugging Path . . . 278
74.2 Common Failures . . . 278
74.3 Logging . . . 278
74.4 Debug the Model Before the Script . . . 278
74.5 Debugging Order . . . 278
74.6 Use Trace Names . . . 278
74.7 Loading and Semantic Diagnostics . . . 279
74.8 Debugging a YAML-Embedded FunctionalP Action . . . 279
74.9 Summary . . . 279

75 FunctionalP Source Debugging in YAML 280
75.1 What Is Supported and What Is Targeted . . . 280
75.2 Design-Time and Runtime Must Stay Separate . . . 280
75.3 Source Provenance Is a First-Class Compilation Product . . . 280
75.4 Breakpoint Identity and State . . . 281
75.5 DAP Source Projection and Runtime Commands . . . 281
75.6 Source Mapping and Incremental Compilation Meet at Identity . . . 282

75.7 Acceptance Scenarios . . . 282

76 Logging, Diagnostics and Traceability 283
76.1 Trace Units . . . 283
76.2 Diagnostics . . . 283
76.3 Traceability . . . 283
76.4 What to Log . . . 283
76.5 Diagnostics for Events . . . 283
76.6 Traceability and Governance . . . 283
76.7 Summary . . . 284

77 Performance, Profiling and Model Complexity 285
77.1 No Benchmark Claim in This Revision . . . 285
77.2 Measure the Whole Path . . . 285
77.3 Model Complexity . . . 286
77.4 Collections and Joins . . . 286
77.5 H-Logic and Hypergraph Queries . . . 286
77.6 Local versus Remote Boundaries . . . 286
77.7 Rich Clients . . . 286
77.8 AI Runtime Measurements . . . 286
77.9 Benchmark Publication Contract . . . 287

78 Extension Patterns and Common Pitfalls 288
78.1 Useful Extension Patterns . . . 288
78.2 Component and Runtime Extension Points . . . 288
78.3 Common Pitfalls . . . 288
78.4 Safe Extension Rule . . . 289
78.5 Summary . . . 289

79 Governing Model Evolution 290
79.1 Change Categories . . . 290
79.2 Compatibility . . . 290
79.3 Documentation . . . 290
79.4 Summary . . . 290

XI Conclusion and Reference 291

80 Runtime Features and Integration Checklist 292
80.1 Core Developer Workflows . . . 292
80.2 Operational Preconditions . . . 292
80.3 Optional and Evolving Capabilities . . . 293
80.4 Maintaining Consistent Documentation . . . 293

81 What Conceptual Programming Changes 294
81.1 From Code to Executable Meaning . . . 294
81.2 From Screens to Projections . . . 294
81.3 From APIs to Capabilities . . . 294
81.4 From AI Prompts to Governed Proposals . . . 294
81.5 From “Supported” to Evidence-Scoped Status . . . 294
81.6 What Remains Conventional Engineering . . . 295
81.7 Summary . . . 295

82 logiCells Project Layout 296
82.1 Scope . . . 296
82.2 Recommended Public Project Shape . . . 296
82.3 What Belongs in the Model Layer . . . 297
82.4 What Belongs in the View Layer . . . 297

83 Conceptual Modeling Quick Reference 298
83.1 Scope . . . 298
83.2 Core Modeling Checklist . . . 298
83.3 New Declarative Module Checklist . . . 299
83.4 Namespace Guidance . . . 299
83.5 Model Smells . . . 299

84 H-Logic Quick Reference 300
84.1 Definitions and Instances . . . 300
84.2 Roles, Attributes and Order . . . 300
84.3 Interfaces and Named Terms . . . 300
84.4 Constraints . . . 300
84.5 Logical Relations . . . 300
84.6 Quantifiers and Queries . . . 301
84.7 Query Modes and Scope . . . 301
84.8 Composite Definitions . . . 301
84.9 Forms to Avoid . . . 302

85 H-Logic Tested Patterns 303
85.1 Primitive Values and Collections . . . 303
85.2 Definitions, Roles and Attributes . . . 303
85.3 Named and Anonymous Instances . . . 303
85.4 Interface Attachment . . . 303
85.5 Quantifiers . . . 304
85.6 Logical Relations and Complements . . . 304
85.7 Composite and Projected Definitions . . . 304
85.8 Commands and Queries . . . 304
85.9 Negative Test Categories . . . 304
85.10Documentation Rule . . . 305

86 Declarative Action Patterns 306
86.1 Scope . . . 306
86.2 Object Actions . . . 306
86.3 Class Actions . . . 306
86.4 Published Actions . . . 306

87 Conceptual Event Patterns 307
87.1 Scope . . . 307
87.2 Event as Observable Change . . . 307
87.3 Local, Visual and Remote Events . . . 307
87.4 Process Events . . . 307

88 Workflow Patterns 308
88.1 Scope . . . 308
88.2 When to Create a Process Model . . . 308
88.3 Synchronous Workflow . . . 308

88.4 Asynchronous Workflow . . . 308
88.5 Anti-Pattern . . . 308

89 AI Assistant Patterns 309
89.1 Scope . . . 309
89.2 Assistant over Concepts . . . 309
89.3 Read, Explain, Act . . . 309
89.4 Guardrails . . . 309

90 Runtime Recipes 310
90.1 Scope . . . 310
90.2 Create an Object . . . 310
90.3 Find a Concept . . . 310
90.4 Execute a Process . . . 310

91 functionalP Language Essentials 311
91.1 Purpose . . . 311
91.2 Program Structure . . . 311
91.3 Sequences . . . 311
91.4 Primitive Values . . . 312
91.5 Assignments and Variables . . . 312
91.6 Arithmetic, Comparisons and Boolean Logic . . . 312
91.7 Lists and Ranges . . . 312
91.8 Control Flow . . . 313
91.9 Functions . . . 313
91.10 Partial Application and Lambdas . . . 314
91.11 Case Expressions . . . 314
91.12 Dot Expressions and Runtime Objects . . . 314
91.13 Hypergraph Literals and Process Constructs . . . 314
91.14 Common Misreadings . . . 314
91.15 Recommended Learning Order . . . 315

92 Glossary of Conceptual Programming 316
92.1 Scope . . . 316
92.2 Selected Terms . . . 316

93 Troubleshooting Guide 317
93.1 Scope . . . 317
93.2 A View Does Not Show the Expected Data . . . 317
93.3 An Action Cannot Be Called Externally . . . 317
93.4 A Process Is Hard to Debug . . . 317
93.5 A Component Is Too Platform-Specific . . . 317

About This Developer Guide

Written for third-party developers and integrators, this guide shows how logiCells turns domain concepts into executable models and reusable application contracts. It teaches the core of declarative conceptual programming through concrete development tasks. Its subject is the executable conceptual model and the runtime that interprets it: concepts, relations, constraints, actions, processes, persistence, publication, testing, diagnostics and governed automation.

The logiCells MVC/View architecture is *not* a prerequisite for using logiCells. A conceptual application can be consumed through a command-line interface, APIs and services, an MVP-style user interface, or another integration boundary. The View architecture is one optional client architecture, intended for richer native or web clients that need local state, cache, asynchronous interaction and the associated client/runtime coordination mechanisms.

For that reason, the detailed View/MVC material has been moved to the companion *logiCells Rich Client View Guide*. This guide only introduces the architectural choice and the publication boundary needed to understand where a client fits.

Runtime platforms and external bindings

The current logiCells runtime is available on Windows, Linux and macOS and can be deployed in containers. Direct ABI integration is currently provided through Python and C# bindings. Platform packaging and host-language bindings are operational projections of the same conceptual model; they do not change the model itself.

How to use this guide

New mechanisms are introduced in this order:

```
business need
-> concept or intention
-> minimal declaration
-> observable result
-> runtime contract and mechanics
```

Listing 1: Learning sequence used in this guide

The reader should be able to explain what the model means before having to explain how a particular UI framework, loader or runtime class implements it.

Names used in examples

Examples use the public/documentation names of runtime types rather than implementation-language class names. When a historical project contains implementation-prefixed class identifiers, treat them as compatibility details of that project, not as vocabulary a new developer must learn.

Part I

Start Here: The Executable Conceptual Model

Chapter 1

The logiCells Mental Model

The easiest way to understand logiCells is to stop thinking first in terms of classes, tables, routes or screens. These elements still exist, but they are not the starting point. The starting point is the conceptual model: what the application knows, how that knowledge is structured, what can be done with it, how it can be observed, and how it can be projected to users, services or agents.

A logiCells application can be read as a network of concepts, instances, relations, facets, actions, events, collections and views. The runtime gives operational meaning to these declarations. The developer therefore works at two levels at once: the conceptual level, where domain meaning is declared, and the runtime level, where those declarations become executable structures.

1.1 Applications as Conceptual Systems

A logiCells model file typically starts with a `class` declaration. The word class should not be read only in the object-oriented sense. In this context, it is the entry point of a conceptual runtime element. It carries a name, a class identifier, a type, optional generator information, object information and then the conceptual sections that define its behavior.

```yaml
class:
  name: UserContact
  classId: lgc#user_contact
  className: CustomClassItem
  currentGeneratorName: Lgc.Common.UserContact
  type: Entity
  generators: []
  object: {}
  concepts: []
  collections: []
  actions: []
  events: []
  views: []
```

Listing 1.1: The same top-level structure in logiCells YAML syntax

This shape gives the mental map of the platform. A model is not just a data structure. It is an organized description of runtime meaning.

1.2 Concepts

A concept defines what kind of thing the model is dealing with. It can declare constraints and facets. A common pattern is to declare an `Entity` concept and state that it implies a more general platform concept such as `lgc#entity`.

```yaml
concepts:
  - Entity:
      constraints:
        - implies: lgc#entity
```

Listing 1.2: Concept and implication in canonical MODEL YAML

The implication does not merely decorate the model. It places the concept in a conceptual hierarchy that the runtime and tools can interpret.

1.3 Facets

Facets separate complementary views of the same concept. A `hypergraph` facet describes conceptual fields. A `database` facet describes persistence mapping. Other facets may be introduced to support specialized projections.

```yaml
facets:
  - main:
      type: hypergraph
      fields:
        - NextExecution: DateTime
        - Executed: Boolean
  - ScheduleActionItem:
      type: database
      map: ScheduleActionItem
      fields:
        - IdScheduleActionItem:
            type: String
            rolePath: Id
        - NextExecution: DateTime
        - Executed: Boolean
```

Listing 1.3: Facets in logiCells YAML syntax

The mental rule is simple: a concept is not reduced to one physical representation. It can have several facets, each one supporting a different runtime concern.

1.4 Relations and Role Paths

Relations often appear through role paths. In the `UserContact` model, fields such as `User.Id`, `User.ClassId`, `Contact.Id` and `Contact.ClassId` show that the model is not only storing flat columns. It is mapping fields to conceptual roles.

```yaml
fields:
  - IdUser:
      type: String
      rolePath: User.Id
      nullable: true
  - ClassIdUser:
      type: String
      rolePath: User.ClassId
      nullable: true
  - IdContact:
      type: String
      rolePath: Contact.Id
      nullable: true
  - ClassIdContact:
      type: String
      rolePath: Contact.ClassId
      nullable: true
```

Listing 1.4: Role paths in logiCells YAML syntax

Role paths are central to the logiCells mental model because they make explicit how storage-level fields correspond to conceptual positions.

1.5 Three Kinds of Values: Scalar, Object Reference and Relation

A frequent source of errors is to treat every JSON, YAML or form value as if it were a scalar. In logiCells, three different contracts must be distinguished early.

Kind	Typical shape	Meaning
Scalar	`Caption: "Ready"`	A value such as String, Integer, Boolean, Float or DateTime.
Typed object reference	`Id + ClassId + FormalName`	A reference to a conceptual object. It must not be reduced to an arbitrary string when the runtime expects an object identity.
Conceptual relation	Named roles / role paths	A fact connecting two or more participants. Storage fields such as `Owner.Id` and `Owner.ClassId` are projections of those roles, not the relation itself.

For a typed reference, the external representation commonly carries the identity triplet explicitly:

```
{
  "Id": "42",
  "ClassId": "lgc#authorized_role",
  "FormalName": "quality-manager"
}
```

Listing 1.5: Typed reference instead of an untyped string

If a published contract expects such a reference, a payload such as `"AuthorizedBy": "quality-manager"` is not an equivalent shorthand unless that shorthand is explicitly documented by the adapter. Treating it as equivalent can produce a false success: the request may be accepted while the relation is not actually resolved.

The same discipline applies to `DateTime`. The publication layer should define one accepted representation and reject malformed values at the request boundary. Do not rely on database or persistence errors to reveal a bad date later in the transaction.

For `ClassId`, use the canonical identifier exactly as published. Case folding, underscore insertion or name fallback must not be assumed unless the contract explicitly declares such normalization.

1.6 Actions

Actions express what can be done with a conceptual element. They should be named as business intentions whenever possible. An action can target a class-level capability or an object-level capability, and it can be published so that other parts of the runtime can call it.

A scheduler manager, for example, may expose a class-level `ScheduleAction`. A scheduled item may expose an object-level `Execute`. The pair expresses an operational chain: the manager finds a pending item, then the item executes its own behavior.

1.7 Events

Events represent observable changes or runtime signals. They make the model reactive. An event does not need to duplicate the whole action logic. Its role is to make a meaningful change visible so that other parts of the system can respond.

In a conceptual runtime, the value of events is not only technical dispatching. Events preserve traceability. They can say which concept changed, which action caused the change, which object was involved and which downstream behavior was triggered.

1.8 Collections and Views

Collections define queryable projections of conceptual structures. Views define how those projections are presented. A collection may be simple, joined, aggregated, backed by an object list or linked to a provider. A view may target a class item, an object item, a list, a form, a dashboard or another publication surface.

The important mental separation is this: collections organize access to conceptual data, while views organize presentation. Both are projections of the model.

1.9 The Runtime

The runtime is the interpreter and executor of the model. It loads declarations, resolves structure, builds runtime elements, exposes collections, routes actions, propagates events and publishes views or services. It should be treated as the execution environment of conceptual declarations.

Implementation technologies remain internal to the runtime. The public model of the book is logiCells itself: concepts, facets, roles, collections, actions, events, views and services.

Chapter 2

First Success: Install, Declare, Validate

This first success is deliberately independent of any UI architecture. Install the current logiCells Community Edition from `logiCells.ai`; installation details are intentionally left to the current distribution. The examples below use the `cell` command-line interface.

2.1 Check the tooling

```
cell doctor
```

Listing 2.1: Check the installed tooling

2.2 Create one complete MODEL source

Create `Task.model.yaml`. The shape below follows the executable declarative grammar: named generators, concepts, facets, fields and parameters use their display-name key; polymorphic children such as actions keep their explicit item type.

```
class:
  classId: lgc#task
  name: Task
  className: CustomClassItem
  type: Entity
  currentGeneratorName: TaskManagement.Task

  generators:
    - TaskManagement.Task:
        nameFieldRecord: Task
        nameField: Title
        class: GUIDStringGeneratorInfo
        params: ''

  object:
    className: CustomObjectItem
    objectName: Task

  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
        facets:
          - main:
              type: hypergraph
              fields:
                - Title: String
                - Status: String
```

```
29          - Task:
30              type: database
31              map: lgcTask
32              fields:
33                - IdTask:
34                    type: String
35                    size: 32
36                    rolePath: Id
37                    generatorName: TaskManagement.Task
38                - ClassIdTask:
39                    type: String
40                    size: 32
41                    rolePath: ClassId
42                    defaultValue: lgc#task
43                - Title:
44                    type: String
45                    size: 200
46                - Status:
47                    type: String
48                    size: 50
49                    defaultValue: Open
```

Listing 2.2: Complete first MODEL source: Task.model.yaml

The names in `className` and generator `class` are runtime binding identifiers required by the current grammar. Documentation uses public binding names and does not expose implementation-language-prefixed class names.

2.3 Validate and inspect

```
1 cell declarative validate Task.model.yaml
```

Listing 2.3: Validate the MODEL source

For automation, add `-json`; stdout then contains one machine-readable command result.

```
1 cell --json declarative validate Task.model.yaml
```

Listing 2.4: Machine-readable validation

Inspect either the declarative file or the MODEL metadata understood by tooling:

```
1 cell declarative inspect Task.model.yaml
2 cell model metadata inspect Task.model.yaml
```

Listing 2.5: Inspect the source

This establishes the core loop used throughout the guide: declare, validate, inspect, then assemble an application or add an interface only when needed.

Chapter 3

Application Interfaces and Architectural Choices

The executable conceptual model is the center of a logiCells application. A client architecture is a way to interact with that model; it is not the definition of the model itself.

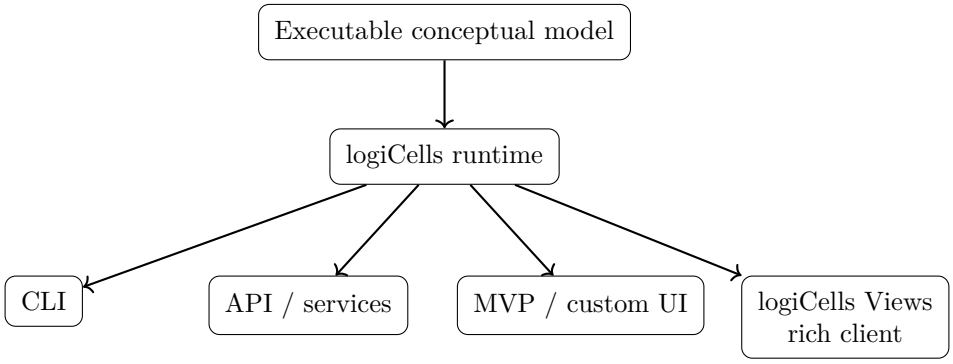


Figure 3.1: Several application interfaces can operate over the same conceptual model.

3.1 CLI

A command-line application can load or address the model and invoke conceptual capabilities without introducing a visual framework. This is useful for tooling, administration, automation, development workflows and headless execution.

3.2 API and services

Published actions can be exposed through service adapters. The route or protocol is a publication surface over the same action contract; it should not become a second definition of the business meaning. OpenAPI and MCP publication are concrete examples; both can be generated from an application/package contract as described in Chapter 61, “Publishing Applications through OpenAPI and MCP”.

3.3 MVP or another custom interface

A user interface may use an MVP-style architecture or another presentation technology while continuing to call model capabilities and observe runtime state. The conceptual model remains independent from that presentation architecture.

3.4 logiCells Views

The View architecture is an optional rich-client architecture. Choose it when the client benefits from logiCells-managed projections, local DataSets/cursors, cache, event routing, asynchronous interaction, native or web rendering, and client/runtime coordination.

Its detailed conventions are deliberately not part of the beginner path in this book. Continue with the companion *logiCells Rich Client View Guide* when these capabilities are required.

3.5 External Host-Language Integration

When an external process needs direct runtime integration rather than a published service, the runtime ABI provides the typed boundary. The bindings currently provided are **Python** and **C#**. Their object-lifetime rules, including borrowed versus caller-owned results, are documented in Chapter 64, “Declarative Models and the Runtime ABI”.

3.6 Decision rule

Choose the simplest interface architecture that satisfies the application requirements. Do not introduce View packages, local DataSets or visual event routing only because they exist in the platform. Introduce them when the client needs the capabilities they provide.

Chapter 4

Documentation, Compatibility and Scope

A reliable logiCells application begins with a clear understanding of its model, runtime version and integration boundaries. This chapter explains how to choose the right reference when implementing or deploying an application.

4.1 Selecting an Authoritative Contract

For the version you deploy, consult the following sources in order:

1. The grammar accepted by the installed compiler and the public runtime API.
2. The generated ABI manifests, language bindings and OpenAPI/MCP descriptors for that build.
3. Version-matched runtime and command-line reference documentation.
4. The conceptual examples and integration patterns in this book.

When an example differs from the interface actually distributed with your runtime, use the versioned interface and report the discrepancy. Do not modify business models to accommodate an outdated tutorial.

4.2 Reading Capability Descriptions

This guide distinguishes *available interfaces*, *application integration patterns* and *planned extensions*. An interface being available does not imply that every operating system, provider, transport or deployment arrangement supports identical behavior. A conceptual pattern illustrates how components fit together; the application remains responsible for its configuration, authorization and operational testing.

4.3 Declarative Grammar and H-Logic

MODEL declarations must follow the grammar recognized by the target compiler. H-Logic definitions and queries likewise follow the parser distributed with that runtime; the forms `define .:\#...` and `ask homomorphic`, `ask unifying` and `ask resolving` are used throughout this book. Treat older syntax shown in unrelated examples as illustrative unless accepted by the installed tools.

4.4 Platform and Integration Boundaries

The runtime, a native module, an interface toolkit and a particular language binding have different compatibility requirements. Check the relevant distribution and its supported host versions before deployment. For effectful API or MCP publications, configure authentication, authorization and any required human confirmation for the actual inbound request path; publication alone never grants permission to execute a business action.

4.5 Before Using an Example in Production

1. Confirm the versions of the compiler, runtime, binding and tooling endpoint.
2. Compile the model using the installed command-line tooling.
3. Exercise normal and rejected requests, including role and session boundaries.
4. Test persistence, retries, recovery and shutdown on the intended host.
5. Record your application-specific decisions and operational limits.

These practices keep the guide useful across releases without turning developer documentation into an internal test report.

Part II

The Conceptual Programming Shift

Chapter 5

Why Declarative Conceptual Programming?

Most business applications are built from concepts, but those concepts are rarely treated as first-class executable material. A user, a contact, an action to execute, a scheduled task, a queue, an import, an export or an onboarding process may appear in many places at once: model metadata, a persistence mapping, a collection, a view, a scriptable action, a service endpoint, a runtime event and an administrative screen. Each representation is useful, but none of them alone should become the only place where the business meaning is hidden.

logiCells starts from a different premise. An application is first described as an executable conceptual model. The model declares concepts, facets, fields, relations, collections, actions, events, views and publication structures. The runtime then materializes those declarations into executable structures. This is why logiCells should not be understood as a simple configuration format. Its declarations are part of the programming model.

This edition uses YAML for MODEL metadata and XML only for canonical `*.view.xml` sources. Legacy MODEL XML may appear in migration discussions but is not a new-development authoring format.

5.1 The Fragmentation Problem

In classical application development, business meaning is often split between several artifacts. A scheduled action, for example, may be represented by a database table, a backend class, a polling service, a script, a log message, a queue entry and a screen used by an administrator. The application works, but the conceptual object called a scheduled action is no longer visible as one coherent unit.

logiCells reduces this fragmentation by allowing a class model to gather several aspects of the same concept. The following YAML declaration shows how a scheduled action item can carry its identity, its hypergraph-facing fields, its database projection and its executable action in the same model.

```yaml
actions:
  - action:
      name: Execute
      type: ObjectItem
      caption: "#Execute"
      published: true
      params:
        - Result:
            dataType: Boolean
            paramType: ptReturn
```

Listing 5.1: Declarative action in canonical MODEL YAML

The action fragment becomes meaningful in the surrounding MODEL declaration. YAML is the canonical authoring notation taught for new MODEL development; legacy MODEL XML is discussed only at migration boundaries. The canonical YAML forms used by this book are summarized in Section 12.2, “How named YAML items are encoded”.

```yaml
class:
  name: ScheduleActionItem
  classId: lgc#schedule_action_item
  className: CustomClassItem
  type: Entity
  generators:
    - Lgc.Common.ScheduleActionItem:
        class: GUIDStringGeneratorInfo
  object:
    className: CustomObjectItem
    objectName: ScheduleActionItem
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
        facets:
          - main:
              type: hypergraph
              fields:
                - NextExecution: DateTime
                - Executed: Boolean
                - Priority: Integer
          - ScheduleActionItem:
              type: database
              map: ScheduleActionItem
              fields:
                - IdScheduleActionItem:
                    type: String
                    generatorName: Lgc.Common.ScheduleActionItem
                    rolePath: Id
                - NextExecution: DateTime
                - Executed: Boolean
                - Priority: Integer
  actions:
    - action:
        name: Execute
        type: ObjectItem
        published: true
        params:
          - Result:
              dataType: Boolean
              paramType: ptReturn
```

Listing 5.2: Equivalent logiCells YAML model projection

The important point is not the syntax. The important point is that the model keeps together several meanings that are often separated: conceptual identity, persistence, runtime fields and executable behavior.

5.2 Why Code-Centric Systems Lose Business Meaning

Code is precise, but code is not always the best place to expose business intention. A method can execute a scheduled item, but the method alone does not say how the item is identified, which conceptual type it belongs to, how it is persisted, how it is exposed in a view, which generator creates its identity, or whether it can be published as a runtime capability.

In a code-centric system, the reader must reconstruct the domain by reading implementation details. In a logiCells model, the reader can start from the declared concepts and then follow their projections: fields, facets, actions, collections, views and events. Code still exists, but it is attached to declared intentions instead of being the only place where intentions are discoverable.

5.3 From Data Models to Conceptual Models

A data model says what can be stored. A conceptual model says what the application is about and how its concepts can be used. A database facet may define fields and mappings, but the concept is broader than the table. It may also have a hypergraph facet, constraints, actions, views and events.

This distinction is visible in the dual use of facets. A `hypergraph` facet expresses conceptual structure. A `database` facet expresses persistence. They are related, but they do not have the same role. logiCells keeps them in the same model so that persistence remains a projection of conceptual meaning rather than an isolated schema.

5.4 Declarative Does Not Mean Passive

The word declarative is sometimes misunderstood. It may suggest static configuration, documentation or diagrams. In logiCells, declarative means executable. A declaration can be loaded, interpreted, instantiated, queried, projected and executed.

An action declaration illustrates this clearly. The model can declare an action named `Execute`, its target level, its parameters and its publication status. The runtime can then expose the action as a capability on the relevant object or class item.

The declaration does not execute by itself in isolation. It becomes executable because the runtime knows how to interpret the action element, its target, its parameters and its associated script or handler.

5.5 logiCells as an Executable Conceptual Platform

logiCells can be understood as a platform for turning conceptual declarations into runtime structures. It provides a conceptual memory, a model structure, action and event mechanisms, collections, views, services and publication mechanisms. These elements are not separate frameworks stitched together after the fact. They are coordinated around the same declared model.

The practical consequence is architectural. The application is no longer first a set of classes, tables, screens and routes. It is first a set of declared concepts and capabilities. Classes, tables, screens and routes become projections or implementation supports.

5.6 When Declarative Conceptual Programming Is Useful

Declarative conceptual programming is especially useful when applications have rich business meaning, many relations, several projections, permissions, workflows, dynamic views, service publication and long-term evolution. It is less useful for one-off scripts or very small applications where the cost of modeling would exceed the benefit.

The approach is particularly relevant for ERP-like systems, operational back offices, workflow platforms, knowledge-driven applications, integration platforms and AI-assisted systems where structured conceptual memory matters.

Chapter 6

Reading Map and Pattern Coherence

This guide is organized around one continuous path: a logiCells application begins as a declarative conceptual model and becomes executable through the runtime. Concepts define the vocabulary, relations connect concepts, facets project them into memory or persistence, actions express capabilities, events make changes observable, process models coordinate work, views project state, services publish capabilities and agents operate on the resulting conceptual surface.

6.1 The Core Chain

```
MODEL
  physical YAML + executable declarative grammar
        |
        | compile / assemble when needed
        v
APPLICATION
  resolved application metadata and active packages
        |
        | semantic compilation / projection when needed
        v
SEMANTIC
  H-Logic sources and compiled conceptual hypergraph
```

Listing 6.1: Three authorities, one application lifecycle

These are related views of the same project, but they are not interchangeable authorities. A file-level question should stay in MODEL. A question about inheritance, active packages or redefinitions belongs to APPLICATION. A question about semantic relations, mappings, queries or H-Logic belongs to SEMANTIC. The rest of the book then shows how actions, events, processes, views, services and agents operate over the appropriate resolved layer.

This distinction keeps the beginner path simple: editing a model does not require understanding the hypergraph engine, while semantic exploration remains available when the application needs explicit conceptual reasoning.

6.2 Platform Patterns and Application Patterns

The namespace `lgc#` is used for platform-level concepts and reusable runtime mechanisms. The namespace `lbs#` may be used for application-level examples. This distinction helps keep public documentation clear: the platform provides reusable mechanisms, while an application uses those mechanisms to model its domain.

6.3 Functional Components and Appearance

A visual model declares functional components. A functional component describes the function expected by the view, such as a field, list, button, grid, frame or dashboard area. The effective component is chosen by registration of a concrete implementation. This means that the conceptual view does not own the final appearance. It owns the functional projection of the concept.

Some components are abstract. They define a function that cannot be instantiated directly. A concrete platform implementation must be registered before the runtime can create an effective component. This rule keeps the model portable across desktop, mobile, web and agent-oriented projections.

6.4 What Holds the Whole Architecture Together

The same conceptual principles apply everywhere: a business object owns local behavior, a relation object gives identity to important connections, a process object coordinates several participants, a controller reacts to events, a published action becomes a service boundary, and messaging enables asynchronous coordination. The remaining chapters detail each of these patterns without exposing client-specific workflows or visual designs.

Chapter 7

Conceptual Programming End-to-End Walkthrough

This chapter walks through a small but realistic logiCells scenario: scheduling and executing runtime actions. The example is intentionally close to existing model files. It involves a `ScheduleActionItem`, which represents an item to execute, and an `ActionSchedulerManager`, which represents the manager capable of finding and executing scheduled work.

The purpose is not to describe every implementation detail. The purpose is to show the chain from conceptual declaration to executable behavior.

7.1 Why a Walkthrough Matters

Conceptual programming can sound abstract if it is presented only through definitions. A walkthrough gives the reader a concrete path: a concept is declared, fields are attached, persistence is mapped, an action is exposed, a collection is queried and runtime behavior is triggered.

The chain we will follow is:

```
concept declaration
-> facets and fields
-> persistence projection
-> action declaration
-> collection access
-> object item selection
-> action execution
-> state update
-> trace and publication
```

Listing 7.1: End-to-end conceptual execution chain

7.2 The Business Scenario

The system must periodically execute pending actions. Each scheduled action item has a next execution date, an execution status and a priority. A scheduler manager reads the pending action queue, selects an item, calls its `Execute` action and marks it as executed.

This scenario is small, but it shows several important concepts: entity modeling, action modeling, runtime collections, object-level behavior and traceability.

7.3 Declaring the Scheduled Item

The scheduled item is declared as an entity. Its conceptual fields are simple: a date, a boolean and a priority.

```yaml
concepts:
  - Entity:
      facets:
        - main:
            type: hypergraph
            fields:
              - NextExecution: DateTime
              - Executed: Boolean
              - Priority: Integer
```

Listing 7.2: Scheduled item fields in logiCells YAML syntax

This is already more than a table definition. The fields are part of a conceptual facet and can therefore be connected to views, actions, filters and persistence projections.

7.4 Adding the Persistence Projection

The database facet maps the conceptual item to persistent fields. It also connects identity fields to role paths such as `Id`, `FormalName` and `ClassId`.

```yaml
concepts:
  - Entity:
      facets:
        - ScheduleActionItem:
            type: database
            map: ScheduleActionItem
            fields:
              - IdScheduleActionItem:
                  type: String
                  generatorName: Lgc.Common.ScheduleActionItem
                  rolePath: Id
              - FormalNameScheduleActionItem:
                  type: String
                  rolePath: FormalName
              - ClassIdScheduleActionItem:
                  type: String
                  defaultValue: lgc#schedule_action_item
                  rolePath: ClassId
              - NextExecution: DateTime
              - Executed: Boolean
              - Priority: Integer
```

Listing 7.3: Database facet in logiCells YAML syntax

The runtime can now understand both the conceptual fields and their persistent representation.

7.5 Declaring the Object-Level Action

The scheduled item exposes an `Execute` action. It returns a boolean result and may be published as a runtime capability.

```yaml
action:
  name: Execute
  class: ObjectItemScriptableObjectElementAction
  type: ObjectItem
  caption: "#Execute"
  published: true
  params:
    - Result:
        dataType: Boolean
        paramType: ptReturn
  scripts:
    - script:
        clientScript: false
```

```
        serverScript: true
        language: functionalP
        classes: Desktop
        cdata: |
          Self.Executed := true;
          Self.PostData();
          Result := true;
```

Listing 7.4: Execute action in logiCells YAML syntax

The action is declared in the model, while its execution is handled by the runtime action mechanism.

7.6 Declaring the Scheduler Manager

The scheduler manager is also declared as an entity. It exposes a class-level action named `ScheduleAction`. The action can retrieve an action queue, open a buffer, select an item, execute it and post the updated state.

```
action:
  name: ScheduleAction
  class: ClassItemScriptableObjectElementAction
  type: ClassItem
  published: true
  params:
    - Result:
        dataType: Boolean
        paramType: ptReturn
  scripts:
    - script:
        clientScript: false
        serverScript: true
        language: functionalP
        classes: Desktop
        cdata: |
      ScheduleActionsCI := Self.ObjectsContainer.ClassItemByClassId('lgc#schedule_actions');
      ScheduleActionsProvider := ScheduleActionsCI.GenerateCollectionProvider('ActionQueue');
      Buffer.Open();
      if not Buffer.Eof() then begin
        lActionItemOI := lScheduleActionsOI.ScheduleActionItem;
        lSuccess := lActionItemOI.Execute();
        lActionItemOI.Executed := true;
        lActionItemOI.PostData();
      end;
```

Listing 7.5: Scheduler manager action outline in logiCells YAML syntax

This is the moment where declarative modeling and executable runtime behavior meet. The model declares the action and the runtime gives it access to the conceptual environment.

7.7 What This Scenario Demonstrates

The scenario demonstrates that logiCells declarations can cover a complete execution path. The model defines the conceptual item, its fields, its identity, its persistence, its executable action and the manager capability that orchestrates execution.

It also shows why the runtime should remain concept-centered. The scheduler does not operate on anonymous records. It operates on conceptual object items that have declared roles, actions and persistence behavior.

7.8 Where to Go Next

The following chapters generalize what this walkthrough introduced. We will distinguish code-centric and concept-centric systems, explain why applications can be treated as executable conceptual models, and show how concepts should be introduced before classes, tables, routes or screens.

Chapter 8

From Code-Centric Systems to Concept-Centric Systems

A code-centric system begins with implementation artifacts: classes, methods, database tables, controllers, components and services. A concept-centric system begins with domain concepts and asks how those concepts are related, constrained, executed, observed and projected. logiCells belongs to this second family.

This does not mean that code disappears. It means that code is no longer the only place where application meaning is located. Code becomes attached to declared conceptual structures.

8.1 The Code-Centric Reflex

When a developer hears "scheduled action", the immediate reflex may be to create a table, a class, a worker service and a method. This is natural, but it can hide the broader conceptual structure. Is the scheduled action an entity? Does it have a conceptual identity? Can it be viewed? Can it be executed through a published action? Can its state be traced? Can another agent reason about it?

logiCells asks these questions before reducing the concept to implementation details.

8.2 Where Business Meaning Goes

In a code-centric implementation, business meaning is distributed across many files. The table knows fields. The class knows properties. The worker knows execution. The UI knows presentation. The API knows payload shape. The documentation knows intention. No single structure expresses the whole meaning.

In logiCells, the model brings these aspects closer together. The same declaration can contain conceptual facets, persistence facets, role paths, actions and views.

```
Code-centric:
  table + class + worker + route + form + documentation

Concept-centric:
  concept + facets + fields + roles + action + view + publication
```

Listing 8.1: Code-centric decomposition versus concept-centric model

8.3 From Objects to Concepts

Objects are useful runtime structures, but a concept is broader than an object. An object often represents an instance with fields and methods. A concept can also carry constraints, relations,

facets, role mappings, events and multiple projections.

The `UserContact` model illustrates this. It is not only an object linking a user and a contact. Its fields map identity and formal names for both roles, and the model can be connected to generators, facets and persistence.

```yaml
fields:
  - IdUser:
      type: String
      rolePath: User.Id
      nullable: true
  - FormalNameUser:
      type: String
      rolePath: User.FormalName
      nullable: true
  - IdContact:
      type: String
      rolePath: Contact.Id
      nullable: true
  - FormalNameContact:
      type: String
      rolePath: Contact.FormalName
      nullable: true
```

Listing 8.2: UserContact role mapping in logiCells YAML syntax

8.4 From Tables to Conceptual Memory

A table stores records. Conceptual memory preserves meaning. logiCells can still use database facets, maps and fields, but those structures should be read as projections of conceptual declarations.

This matters for evolution. If a table is the source of truth, every change is first a storage change. If a concept is the source of truth, storage is adapted to preserve and expose conceptual meaning.

8.5 From Routes to Capabilities

A route is a technical entry point. A capability is a conceptual intention that can be published through one or more technical channels. In logiCells, an action such as `Execute` or `ScheduleAction` should be understood first as a capability. Whether it is called from a UI, a service, a script or an agent is a publication concern.

8.6 From Screens to Views

A screen is a concrete user interface. A view is a projection of the conceptual model. The distinction is important because the same conceptual view may be published differently depending on context: as a form, a list, a dashboard, a web component or a service representation.

8.7 Consequences for Architecture

A concept-centric architecture changes the order of design. One first names the concept, its identity, its roles, its constraints, its actions and its projections. Only after that should one decide how it is stored, rendered or exposed.

The result is not less technical. It is technically disciplined around a visible conceptual core.

Chapter 9

Applications as Executable Conceptual Models

A logiCells application is not only a program that manipulates data. It is an executable conceptual model. The model contains declarations that the runtime can interpret: concepts, facets, fields, constraints, collections, actions, events and views.

The word executable is essential. A model is executable when it can produce runtime objects, runtime fields, collections, action calls, events, views and service capabilities. The model is not only an analysis artifact; it participates in the running system.

9.1 From Application as Program to Application as Model

In a traditional architecture, an application is often understood as a body of code organized into layers. In logiCells, the application can be understood as a conceptual model interpreted by a runtime. Code is still present, especially in scripts and handlers, but the surrounding structure is declared.

A scheduled action application, for example, is not only a worker loop. It is a set of concepts: scheduled item, queue, manager, action, execution status and priority.

9.2 What Makes a Model Executable

A model becomes executable when the runtime can answer operational questions from it:

- What is the concept identifier?
- How is identity generated?
- Which fields exist and what are their types?
- Which fields correspond to conceptual roles?
- Which persistence projection should be used?
- Which actions are available?
- Which views or collections can be published?

The following declarative MODEL already contains several of these answers.

```yaml
class:
  name: ScheduleActionItem
  classId: lgc#schedule_action_item
  currentGeneratorName: Lgc.Common.ScheduleActionItem
  type: Entity
  generators:
    - class: GUIDStringGeneratorInfo
      generatorName: Lgc.Common.ScheduleActionItem
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
```

Listing 9.1: Executable model information in logiCells YAML syntax

9.3 The Application Boundary

The application boundary is not only a source-code boundary. It is also a conceptual boundary. It defines which concepts belong to the application, which capabilities are published, which views are available and which external systems interact with it.

For example, the scheduler boundary may include `ScheduleActionItem`, `ScheduleActions`, `ActionSchedulerManager` and the collections used to select pending items. It may expose only a small number of actions externally, while keeping implementation-specific scripts internal.

9.4 Actions as Part of the Model

Actions are not afterthoughts added once the data structure is complete. They are part of the model because they express what the application can do. An executable conceptual model should therefore include action declarations close to the concepts they operate on.

9.5 Views as Optional Projections

A user interface can project the same conceptual model, but the logiCells View architecture is not required to make a model executable. A CLI, an API, a custom MVP client or another adapter can use the same declared concepts and actions. When a cached rich client is needed, the companion *logiCells Rich Client View Guide* defines the canonical `*.view.xml` contract and its DataSet/event lifecycle.

The important rule for this guide is that a view is never the concept itself. It is one optional interaction projection of the same model.

9.6 Why Executability Matters

Executability matters because it makes the model testable, inspectable and governable. A passive diagram can be outdated without immediately breaking anything. An executable model is continuously confronted with runtime behavior. If an action, field, view or collection is wrong, the inconsistency becomes operationally visible.

Chapter 10

Concepts before Classes, Tables and Routes

logiCells encourages a simple discipline: name and structure the concept before choosing the class, the table, the route or the screen. This does not deny the importance of technical artifacts. It prevents them from becoming the first and only source of meaning.

10.1 The Technical Reflex

When a requirement arrives, technical design often starts immediately: create a class, create a table, create an endpoint, create a form. This reflex is efficient for small features, but it can damage long-term conceptual coherence.

For example, the requirement "link a user to a contact" could be implemented as a join table. In logiCells, it is first a concept or relation: `UserContact`. Only then does one define how it is generated, persisted, viewed and used.

10.2 The Conceptual Order

The conceptual order is:

```
concept
-> identity
-> roles and relations
-> constraints
-> facets
-> actions
-> collections and views
-> storage and publication
```

Listing 10.1: Conceptual design order

This order keeps the business structure visible.

10.3 Concepts before Classes

The `class:` declaration in YAML is necessary, but the conceptual question comes first: what is being modeled? In the following example, `UserContact` is a relation-like entity between a user and a contact. Its technical class support is secondary.

```
class:
  name: UserContact
  classId: lgc#user_contact
  type: Entity
```

```
  implementation:
    className: CustomClassItem
    objectClassName: CustomObjectItem
    objectName: UserContact
```

Listing 10.2: UserContact as logiCells YAML concept model

The YAML projection deliberately separates the conceptual name from implementation support.

10.4 Concepts before Tables

A database table can persist `UserContact`, but the table does not define the whole meaning. The role paths show that the fields are linked to two conceptual participants.

```
fields:
  - IdUser:
      type: String
      rolePath: User.Id
  - ClassIdUser:
      type: String
      rolePath: User.ClassId
  - IdContact:
      type: String
      rolePath: Contact.Id
  - ClassIdContact:
      type: String
      rolePath: Contact.ClassId
```

Listing 10.3: Role projections in logiCells YAML syntax

The persistence structure becomes a projection of the conceptual relation.

10.5 Concepts before Routes

A route exposes a capability through a network protocol. It should not define the capability itself. If the model declares `Execute` or `ScheduleAction`, those names express conceptual intentions. A route may later expose them, but the route is only one publication form.

10.6 Concepts before Screens

A screen presents a concept to a user. It should not become the definition of the concept. In logiCells, view declarations should be understood as projections of conceptual structures. This helps the same concept be rendered differently in different contexts without duplicating its meaning.

10.7 Why This Discipline Matters

This discipline matters because business systems live for a long time. Tables are refactored, screens are redesigned, services are replaced and implementations evolve. If the concept remains stable and explicit, the system can evolve without losing its meaning.

Chapter 11

Metadata, Models and Runtime Meaning

Metadata is often treated as secondary information: labels, configuration options, schema hints or UI captions. In logiCells, metadata has a stronger role. It describes structures that the runtime can interpret and execute. This is why one should speak of executable metadata.

The declarative structure contract makes this explicit. It defines declarative elements, attributes, file types and rule-resolution mechanisms. The platform knows which children are valid under which parent rules, which attributes are required and how a declaration must be interpreted.

11.1 Metadata Is Not Decoration

A model attribute such as `classId` is not decorative. It identifies a conceptual runtime class. A `rolePath` is not decorative. It connects a field to a conceptual role. A `published` flag on an action is not decorative. It participates in whether a capability is visible to the runtime environment.

logiCells metadata should therefore be read as runtime-significant information.

11.2 Declarative Structure

A declarative model has rules. At the top level, a model file can declare a `class`. Under a class, the model can contain sections such as `generators`, `object`, `concepts`, `collections`, `events`, `actions` and `views`. Under a concept, the model can contain `constraints` and `facets`. Under a facet, the model can contain fields.

```yaml
class:
  generators: []
  object: {}
  concepts:
    - Entity:
        constraints: []
        facets: []
  collections: []
  events: []
  actions: []
  views: []
```

Listing 11.1: Equivalent logiCells YAML structure

The declarative tooling can use this structure immediately at the MODEL layer to validate, navigate and mutate the source. Application materialization is a later step when the same declarations are assembled for execution.

11.3 Three Authorities for Three Questions

A useful developer habit is to ask which question is being answered before choosing a tool:

MODEL What is physically declared in this YAML file? The source file and executable declarative grammar are authoritative.

APPLICATION What is active after packages, inheritance, clones and redefinitions have been assembled? The resolved application metadata are authoritative.

SEMANTIC What conceptual relation, H-Logic pattern or semantic graph is valid? The H-Logic source and compiled semantic graph are authoritative according to the operation.

A MODEL edit does not need to initialize an application or hypergraph. Conversely, an APPLICATION question should not be answered by merely rereading one file when other packages or redefinitions may change the effective result.

11.4 From Descriptive Metadata to Executable Metadata

Metadata becomes executable when runtime components can act on it. A field can become a property. A facet can become a projection. A collection can become a provider. An action can become a callable capability. A view can become a presentation structure.

This is why the same declarative field can be read at several levels. A `field` is a line in a model file, but it is also a typed declaration, a possible database column, a conceptual role projection and a potential UI or service field.

11.5 Model Meaning and Runtime Meaning

Model meaning is what the declaration expresses for humans. Runtime meaning is what the platform can do with that declaration. logiCells connects the two by making declarative elements and attributes explicit.

For example:

```
ClassIdScheduleActionItem:
  type: String
  defaultValue: lgc#schedule_action_item
  caption: ClassId
  rolePath: ClassId
```

Listing 11.2: Field metadata in logiCells YAML syntax

The human reader sees an identity-related field. The runtime sees a typed field with a default value and a role path.

11.6 Conceptual Metadata versus Technical Metadata

Conceptual metadata expresses domain and runtime meaning: concept names, role paths, constraints, facets, actions and views. Technical metadata expresses implementation support: class names, generator classes, script classes, physical maps and paths.

Both are necessary. The rule for this book is to foreground conceptual metadata and explain technical metadata only when it clarifies runtime execution.

11.7 YAML MODEL Source and the Declarative Contract

YAML is the canonical MODEL authoring syntax in this guide. The executable declarative grammar defines the source contract. Legacy MODEL XML can be accepted at migration boundaries when the runtime compatibility layer supports it, but new modules should not produce a parallel XML MODEL representation. VIEW XML is a separate visual-source contract described in the companion *logiCells Rich Client View Guide*.

11.8 A Model Is Not a Diagram

A diagram can explain a model, but it does not necessarily define a runtime structure. A logiCells model must contain enough formal information to be interpreted: elements, attributes, required fields, allowed children and executable declarations.

This is why the declarative structure matters. It gives the runtime a contract for reading models.

Chapter 12

The Executable Declarative Grammar

The YAML authoring language is not an informal convention. The current framework registers executable declarative structures by file type. Each structure defines legal parent/child relationships, list behavior, display-name mapping, required attributes and defaults. The tooling and runtime consume declarations according to these registered rules; internal representation is deliberately outside this guide's contract.

12.1 Supported declarative file families

The current grammar distinguishes application, package, manifest, model, multi-model container, view, service, IA, translation-map and deployment file types. A `.models` file is only a physical container: classes inside it use exactly the same MODEL class grammar as a standalone `.model` source.

12.2 How named YAML items are encoded

A rule can expose an element name from the tag itself or from one of its attributes. This is why canonical YAML frequently uses a named map item rather than `name:` inside the object.

```yaml
generators:
  - TaskManagement.Task:
      nameField: Title

concepts:
  - Entity:
      facets:
        - main:
            type: hypergraph
            fields:
              - Title: String
```

Listing 12.1: Canonical display-name forms

Here the keys `TaskManagement.Task`, `Entity`, `main` and `Title` carry attributes selected by the grammar (for example generator name, concept name, facet name or field name). Do not replace these with a generic `- name: ...` form unless that exact parent rule declares such a shape.

12.3 MODEL root contract

A MODEL class currently requires `classId`, `name`, `className` and `type`. Optional root attributes include `currentGeneratorName`, `redefined` and `namespace`. The same class rule is reused in standalone MODEL files, legacy view-behind locations and `.models` containers.

12.4 Canonical actions

Under `actions`, `action` and `actionGroup` are polymorphic children, so the explicit item tag remains visible. A direct action requires `name` and `type`; the current grammar also recognizes optional `class`, `path`, `caption`, `published`, relay, parameter-view, dynamic-parameter, mode and inheritance attributes.

```yaml
actions:
  - action:
      name: Execute
      type: ObjectItem
      published: true
      params:
        - Result:
            dataType: Boolean
            paramType: ptReturn
      scripts:
        - script:
            clientScript: false
            serverScript: true
            language: functionalP
            cdata: |
              Self.Status := 'Executed';
              Result := true;
```

Listing 12.2: Canonical direct action with typed result and script

Action parameters require `dataType` and `paramType`; `defaultValue` is optional. Action scripts require `clientScript`, `serverScript` and `language`; `classes` and `cdata` are optional in the grammar.

12.5 Events are not action declarations

An event action entry is a relay/binding to an action. Its grammar is different from an action declaration. Redirect parameters use a small binding grammar:

```
name        target parameter name
eventParam  value forwarded from the source event
value       literal constant assigned by the redirect
path        value resolved through a path
```

Listing 12.3: Event-action redirect parameter semantics

`value` is a literal value, not executable code. Script attributes such as `scripted` or `language` do not belong to this redirect-parameter grammar. Use `eventParam` when the value comes from the event and `value` when the redirect supplies a constant. Do not copy the direct-action parameter declaration grammar (`dataType`, `paramType`, `defaultValue`) into an event relay.

12.6 Collections and role paths

Collection families include `collection`, `objectItemListCollection`, `storedProcedure`, `agregate`, `union` and `join`. Field rules expose both physical metadata (field name/type/size) and conceptual mapping such as `rolePath`. Keep this distinction explicit: the collection or database projection may change while the role path preserves the conceptual meaning.

12.7 Compatibility versus canonical authoring

The runtime contains historical compatibility paths, including legacy XML forms. New documentation and generated YAML must follow the registered declarative grammar rather than reproducing permissive historical examples. When a source example conflicts with the registered rules, the registered structure is the authority for authoring syntax; runtime compatibility is documented separately.

12.8 Public names versus implementation bindings

Some grammar attributes select runtime bindings, notably `className` or `class`. These attributes are part of the current metadata contract, but the guide uses public registry/binding names and deliberately omits implementation-language-prefixed names. A runtime implementation identifier is not domain vocabulary.

Chapter 13

Migrating Existing XML Metadata to YAML

Migration is not a text-format conversion exercise. The goal is to preserve runtime meaning while moving historical XML metadata into the current logiCells YAML syntax and canonical declarative structure.

13.1 The Migration Pipeline

A reliable migration uses a round-trip and a runtime test:

```
historical XML
    -> migration tooling
    -> canonical YAML
    -> semantic comparison with historical source
    -> runtime load
    -> model/application tests
```

Listing 13.1: Recommended migration workflow

Migration validation should focus on runtime-significant semantics rather than on internal serialization steps. Canonical YAML is the target authoring form; verification compares the resulting model behavior and metadata meaning with the legacy source.

13.2 Compare Meaning, Not Formatting

A byte-for-byte XML diff is too strict because the canonical grammar can introduce wrappers or normalize representation. Compare runtime-significant content instead:

- class and package identity (especially `classId`);
- concepts, facets and constraints;
- field names, types, role paths and persistence properties;
- states and state indexes;
- collections, headers and embedded object data;
- actions, action groups, parameters and scripts;
- events and views;
- service definitions and publication metadata.

13.3 Known Migration Hazards

13.3.1 Hash-prefixed values

Quote values such as `'#top'` so that YAML does not treat them as comments.

13.3.2 Compact versus expanded items

A compact item such as `- Count: Integer` encodes both a display name and a display value. When XML2Yaml expands that item to add properties, it must preserve the display value (for fields, the `type`).

13.3.3 Polymorphic sequences

Do not infer sibling types when the list may contain different element kinds. Use explicit keys such as `action:` and `actionGroup:`.

13.3.4 Historical wrappers

The canonical declarative form may contain containers such as `classes`, `items`, `params` or `actions` even when a historical XML source also uses a shorthand without them. The canonical form should be preferred in newly generated metadata.

13.3.5 Embedded collection data

Headers and object rows are metadata, not arbitrary YAML. Preserve field names, `dataType`, sizes, role paths and row values explicitly.

13.4 Migrating Legacy Views to Interaction Contracts

Legacy `.form`, `.frame` and `.view` sources should migrate toward a mixed contract representation rather than toward an all-YAML screen description:

```
meta/views/
  contracts/
    Common/Folder/
      Fields.view.xml
      List.view.xml
      List.models.yaml
      _.manifest.yaml
  models/
    Common/Folder/
      BusinessClass.model.yaml
      _.manifest.yaml
```

Listing 13.2: Canonical migrated VIEW tree

The `contracts` tree contains channel-independent interaction contracts. A contract may still contain visual layout, but it can also be consumed by an agentic runtime that presents information and requests missing values without rendering a form. Inline MODEL/page-model/component-model declarations are externalized to colocated `*.models.yaml` only when they exist. Do not generate an empty models companion for a contract with no MODEL content.

Each externalized class is referenced from the XML contract with `source`, `class` and `classId`. The local `_.manifest.yaml` uses `fileName` to point to the physical `*.models.yaml` container. Business-class view metadata that must be separated from the business package is represented as a minimal redefinition under `meta/views/models`; do not duplicate the entire business class.

The migration tool should also avoid reproducing the old source directory tree directly under `meta/views`. The canonical immediate children are `contracts` and `models`.

13.5 Three Levels of Validation

A migration should pass three increasingly strong checks:

1. **Syntax**: YAML parses and XML is well formed.
2. **Grammar**: keys and elements are valid for their declarative parent and file type.
3. **Runtime**: manifests load, models compile, metadata completion succeeds and the application reaches its expected initialized state.

A syntactically valid YAML file can still be semantically incomplete. Runtime loading is therefore part of migration validation, not an optional final polish.

13.6 What Not to Migrate Automatically

Not every historical anomaly deserves a new grammar rule. If an old test file uses a structure that no supported runtime model requires, clean or retire the source instead of adding permanent ambiguity. Compatibility should be intentional, localized and justified by real deployed metadata.

13.7 Round-trip Regression Tests

For model families that matter to production, retain representative XML/YAML fixtures and compare their runtime-significant declarative structure. Include at least compact fields, expanded fields, named mappings, nested action groups, states, constraints with hash-prefixed values, collection data and service definitions. These fixtures make future codec optimization safer because performance changes can be checked against stable semantics.

Part III

Conceptual Modeling Foundations

Chapter 14

Concepts, Instances and Relations

A logiCells application begins with concepts. A concept describes a kind of thing that exists in the application domain or in the platform itself. An instance is a concrete occurrence of that concept. A relation connects concepts or instances and gives structure to the model. Together, concepts, instances and relations form the basic vocabulary of declarative conceptual programming.

The distinction is simple, but it is fundamental. A class, a table, a form or an endpoint may later be produced from the model. They are not the first level of meaning. The first level of meaning is conceptual: what kinds of things exist, how they are identified, how they relate, what constraints apply to them, and what actions can be performed on them.

14.1 Concepts

A concept defines a type of meaning. In current YAML MODEL files, a conceptual class is declared under the `class:` root. The `name` gives the model name, the `classId` gives the stable conceptual identifier, and the `type` indicates the broad category of the model element.

```yaml
facets:
  - main:
      type: hypergraph
      fields:
        - NextExecution: DateTime
        - Executed: Boolean
        - Priority: Integer
  - ScheduleActionItem:
      type: database
      map: ScheduleActionItem
      fields:
        - IdScheduleActionItem:
            type: String
            rolePath: Id
        - ClassIdScheduleActionItem:
            type: String
            defaultValue: lgc#schedule_action_item
            rolePath: ClassId
```

Listing 14.1: Facets in logiCells YAML syntax

In this example, `ScheduleActionItem` is declared as an entity-like concept. The constraint says that this concept implies the more general platform concept `lgc#entity`. The model therefore does not only introduce a technical object; it places the object in a conceptual hierarchy.

```yaml
class:
  name: ScheduleActionItem
  classId: lgc#schedule_action_item
  type: Entity
  concepts:
    - Entity:
```

```
7         constraints:
8           - implies: lgc#entity
```

Listing 14.2: Concept declaration in logiCells YAML syntax

14.2 Instances

An instance is a concrete object that conforms to a concept. The model may declare how such objects are represented and generated. In YAML, the `object:` declaration identifies the runtime object representation associated with the class.

```
1 object:
2   className: CustomObjectItem
3   objectName: ScheduleActionItem
```

Listing 14.3: Object representation in logiCells YAML syntax

In conceptual terms, this line means that the model has an instance-level representation. The book should not present this as the main abstraction; it is an implementation-facing declaration. The public idea is that the concept can have runtime instances, and that those instances can carry fields, participate in relations, be persisted, displayed and acted upon.

14.3 Conceptual Identifiers

A concept needs a stable identity. In the examples, `lbs#schedule_action_item`, `lbs#user_contact` and `lgc#user` are conceptual identifiers. They are not display labels. They are stable names used by the runtime to recognize the conceptual type.

This is different from a class name or a table name. A class name can change because of refactoring. A table name can change because of persistence design. A conceptual identifier should remain stable because it is part of the model's semantic identity.

14.4 Facets

A concept can have facets. A facet is a projection of the concept for a specific runtime concern. In current YAML models, one often sees a `hypergraph` facet and a `database` facet. The hypergraph facet expresses conceptual/runtime fields. The database facet maps conceptual information to persistent records.

The important point is not that logiCells has several storage formats. The important point is that the same concept can have several projections without losing its identity. The hypergraph facet and the database facet are different views of the same conceptual object.

14.5 Relations

A relation connects concepts or instances. Sometimes relations are explicit conceptual objects. Sometimes they appear through role paths inside a facet. The `UserContact` model is a good example. It connects a user and a contact through fields whose `rolePath` values refer to `User.Id`, `User.ClassId`, `Contact.Id` and `Contact.ClassId`.

```
1 fields:
2   - IdUser:
3       type: String
4       rolePath: User.Id
5       nullable: true
6   - ClassIdUser:
7       type: String
```

```
      defaultValue: lgc#user
      rolePath: User.ClassId
      nullable: true
  - IdContact:
      type: String
      rolePath: Contact.Id
      nullable: true
  - ClassIdContact:
      type: String
      defaultValue: lgc#contact
      rolePath: Contact.ClassId
      nullable: true
```

Listing 14.4: The same relation in logiCells YAML syntax

A traditional database reading would see foreign-key-like fields. A conceptual reading sees roles: this object carries a `User` role and a `Contact` role. The database facet is only one way to persist that relation.

14.6 Constraints

Constraints define what must be true for a concept, a role or a field. They are not only validation rules at the edge of the system. They are part of the model's meaning. In the simple examples above, the implication constraint places a concept under a more general platform concept. Other constraints can express defaults, fixed values, non-nullity, role domains and contextual validity.

A constraint should be read as a conceptual commitment. It tells the runtime, the tools and the reader how the model is meant to behave.

14.7 Concepts, Instances and Relations in Execution

Concepts are not passive documentation. Once interpreted by the runtime, they can be used to create instances, resolve collections, invoke actions, publish views and expose capabilities. The scheduler example illustrates this principle. `ScheduleActionItem` is a concept. A particular scheduled item is an instance. The scheduler manager uses a collection to retrieve instances, invokes their `Execute` action, and updates their `Executed` state.

This is the point at which declarative conceptual programming becomes operational. The model is not a diagram placed beside the application. It participates in the application.

Chapter 15

Types, Roles and Polyadic Relations

logiCells does not reduce relations to anonymous links between two nodes. A relation may have a type, named participants, constraints on each participant, and an identity of its own. This is essential for declarative conceptual programming, because many business facts are not simple properties. They connect several participants at once and each participant has a specific meaning inside the fact.

A scheduled action, for example, is not only an object with a date. It can relate an action definition, a scheduling rule, a queue, an execution context, a target object and an execution result. Treating this structure as a set of independent technical fields loses meaning. Treating it as a typed conceptual structure preserves the intention of the model.

15.1 From Fields to Conceptual Roles

In the logiCells model files used throughout this book, relation participants are usually represented by fields in a conceptual facet and by `rolePath` projections in a persistence facet. The declarative MODEL notation does not normally introduce a dedicated role element for business relation participants. Instead, the participant role is carried by the field name, its type, and the role path used when the relation is projected.

A field named `IdUser` only says that an identifier is stored somewhere. A conceptual field named `User`, projected through `User.Id` and `User.ClassId`, says that the current structure carries a participant playing the role of user. This difference is important: the conceptual role is defined in the model, while the database fields are only one projection of that role.

```
class:
  name: ScheduleActionItem
  classId: lgc#schedule_action_item
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
        facets:
          - main:
              type: hypergraph
              fields:
                - Action: lgc#action
                - Queue: lgc#queue
                - NextExecution: DateTime
                - Executed: Boolean
          - ScheduleActionItem:
              type: database
              map: lgcScheduleActionItem
              fields:
                - IdAction:
                    type: String
                    rolePath: Action.Id
```

```
                - ClassIdAction:
                    type: String
                    rolePath: Action.ClassId
                    defaultValue: lgc#action
                - IdQueue:
                    type: String
                    rolePath: Queue.Id
                - ClassIdQueue:
                    type: String
                    rolePath: Queue.ClassId
                    defaultValue: lgc#queue
                - NextExecution: DateTime
                - Executed: Boolean
```

Listing 15.1: Typed participants in YAML model notation

YAML is the native declarative model syntax used to describe logiCells business models. Its structure is aligned with the declarative metadata contracts implemented by the logiCells runtime. XML is another representation of the same modeling intentions.

15.2 Role Names Are Part of the Model

A participant name is not cosmetic. It is a piece of model semantics. If two participants have the same type but different meanings, they should not be collapsed into a single unnamed reference. In a contact model, for example, a user may be the owner of a contact, the creator of the contact, the last person who modified it, or the person represented by the contact. These references may all point to user-like concepts, but they are not interchangeable.

Role names also make generated views, services and AI explanations more stable. A model that declares a participant named `Queue` can later explain why an action is waiting. A model that only stores a numeric queue identifier cannot provide the same conceptual interpretation without external documentation.

15.3 Binary Relations

Binary relations connect two participants. They are useful for common cases: a user owns a contact, an action belongs to a queue, a category classifies an item, or a scheduled item triggers an action.

In logiCells model files, such a relation is normally represented as a conceptual class or relation model. Its participants are declared as fields in a conceptual facet and projected through `rolePath` fields in the database facet.

```
class:
  name: BelongsToQueue
  classId: lgc#belongs_to_queue
  className: CustomClassItem
  type: Relation
  concepts:
    - Relation:
        constraints:
          - implies: lgc#relation
        facets:
          - main:
              type: hypergraph
              fields:
                - Item: lgc#schedule_action_item
                - Queue: lgc#queue
          - BelongsToQueue:
              type: database
              map: lgcBelongsToQueue
              fields:
                - IdBelongsToQueue:
                    type: String
```

```
                    rolePath: Id
                - ClassIdBelongsToQueue:
                    type: String
                    rolePath: ClassId
                    defaultValue: lgc#belongs_to_queue
                - IdItem:
                    type: String
                    rolePath: Item.Id
                - ClassIdItem:
                    type: String
                    rolePath: Item.ClassId
                    defaultValue: lgc#schedule_action_item
                - IdQueue:
                    type: String
                    rolePath: Queue.Id
                - ClassIdQueue:
                    type: String
                    rolePath: Queue.ClassId
                    defaultValue: lgc#queue
```

Listing 15.2: The same binary relation in YAML model notation

A binary relation is still a conceptual object. It may have constraints, metadata, visibility rules, persistence projections and events attached to it.

15.4 Polyadic Relations

A polyadic relation connects more than two participants. This is where logiCells departs from graph models that force every fact into a binary edge. Business facts often have three, four or more participants. Forcing such facts into chains of binary links may work technically, but it can hide the actual conceptual unit.

Consider the execution of a scheduled action. The relevant fact may involve the scheduled item, the action definition, the queue, the execution context, the execution time and the result. This is one business event, not a random collection of unrelated links.

```
class:
  name: ScheduledExecution
  classId: lgc#scheduled_execution
  className: CustomClassItem
  type: Relation
  concepts:
    - Relation:
        constraints:
          - implies: lgc#relation
        facets:
          - main:
              type: hypergraph
              fields:
                - Item: lgc#schedule_action_item
                - Action: lgc#action
                - Queue: lgc#queue
                - Context: lgc#execution_context
                - PlannedTime: DateTime
                - Result:
                    type: lgc#action_result
                    nullable: true
          - ScheduledExecution:
              type: database
              map: lgcScheduledExecution
              fields:
                - IdScheduledExecution:
                    type: String
                    rolePath: Id
                - ClassIdScheduledExecution:
                    type: String
                    rolePath: ClassId
                    defaultValue: lgc#scheduled_execution
                - IdItem:
```

```
                    type: String
                    rolePath: Item.Id
                - IdAction:
                    type: String
                    rolePath: Action.Id
                - IdQueue:
                    type: String
                    rolePath: Queue.Id
                - IdContext:
                    type: String
                    rolePath: Context.Id
                - PlannedTime: DateTime
                - IdResult:
                    type: String
                    rolePath: Result.Id
                    nullable: true
```

Listing 15.3: The same polyadic relation in YAML model notation

The conceptual advantage is that the execution fact can now be named, inspected, constrained, queried and explained as a whole.

15.5 Relation Identity

A relation may need its own identity. This is especially true when the relation is not merely structural but also historical, auditable or actionable. A contact assignment, a scheduled execution, a validation decision or an import/export operation can become an entity-like fact with timestamps, status, provenance and events.

In logiCells, this is where the distinction between a relation and a reified fact becomes important. The relation describes the structure. The reified fact gives an occurrence of that structure a stable runtime identity.

```
class:
  name: ScheduledExecutionFact
  classId: lgc#scheduled_execution_fact
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: lgc#scheduled_execution
        facets:
          - main:
              type: hypergraph
              fields:
                - Item: lgc#schedule_action_item
                - Action: lgc#action
                - Queue: lgc#queue
                - PlannedTime: DateTime
                - ExecutedAt: DateTime
                - Status: String
```

Listing 15.4: A reified relation occurrence in YAML model notation

15.6 Polyadic Maps

A polyadic map expresses how participants in one conceptual structure align with participants or parameters in another. It is useful when a model is transformed, specialized, projected or executed. For example, a scheduled item may be projected into a service call, or an action declaration may be projected into an executable runtime command.

The point is not to introduce another technical mapping layer. The point is to keep the alignment explicit. If the source participant `PlannedTime` becomes the target parameter `StartAt`, the model should be able to say so.

```yaml
polyadicMap:
  name: scheduleItemToActionCall
  source: lgc#scheduled_execution
  target: lgc#action_call
  mapping:
    Item: TargetObject
    Action: Operation
    PlannedTime: StartAt
    Context: ExecutionContext
```

Listing 15.5: Participant alignment in YAML model notation

15.6.1 Explicit Maps, Derived Maps and Non-applicable Constraints

A mapping is evidence that roles from two conceptual structures correspond. When the model provides an explicit map, that declaration has priority. When logiCells derives a map, it may only use role correspondences that are actually compatible in the source and target structures.

For example, if a constraint expression exposes the roles `Amount` and `Limit`, it can be bound to a source concept that exposes compatible roles with those names. If the source instead exposes only `Country`, there is no meaningful inferred mapping. The correct result is an inapplicable constraint, not a positional guess.

```
explicit mapping exists
  -> use it

otherwise
  -> align compatible source and target roles

no compatible roles
  -> no valid polyadic map
  -> constraint is not applicable to that source
```

Listing 15.6: Conceptual rule for inferred mappings

Do not invent synthetic roles merely to make a relation fit. This rule keeps a polyadic map semantic: it records a correspondence that exists in the model instead of hiding a mismatch.

15.7 Design Rules

Model a relation as a conceptual structure when the link has business meaning. Use fields in the hypergraph facet to name the participants. Use `rolePath` in persistence facets to project participant identities and class identifiers. Give the relation its own model when it must be queried, constrained, audited, displayed, published or used by an action. Do not introduce a synthetic XML role element for relation participants unless the actual model grammar defines one for that purpose. In the examples used by this guide, business relation participants are carried by fields, facets, joins, collections, constraints and `rolePath` projections.

Chapter 16

Identity, Context and Conceptual Scope

A conceptual model needs stable identity. Without identity, the runtime cannot reliably distinguish between a concept, an instance, a relation occurrence, a projection, or a contextual view of the same underlying object. logiCells therefore treats identity and context as central parts of declarative conceptual programming.

Identity answers the question: what is this thing? Context answers the question: under which conceptual conditions is this thing valid, visible or interpreted? Scope answers the question: where does a declaration apply?

16.1 Conceptual Identity

A concept has a stable conceptual identifier. In YAML MODEL files this appears through properties such as `classId`, `relationId`, or other model identifiers. The identifier should not be confused with a database primary key. A database key identifies a persisted row. A conceptual identifier identifies a meaning in the model.

```yaml
class:
  name: UserContact
  classId: lgc#user_contact
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
```

Listing 16.1: Conceptual identity in logiCells YAML syntax

The identifier `lbs#user_contact` gives the model element a stable conceptual name. The runtime may later create instances, views, actions and persistence projections from this declaration, but the conceptual identity remains the anchor.

16.2 Instance Identity

Instance identity belongs to concrete occurrences. Two contacts may share the same name, the same email domain or the same category, but they remain distinct instances. Conversely, the same instance may be shown in several views, exported through services, enriched with additional facets or placed in different contexts.

A logiCells model should therefore avoid confusing display labels with identity. A label can change. A conceptual instance identity should remain stable enough for persistence, traceability, synchronization and event routing.

```yaml
instance:
  id: contact-00042
  type: lgc#user_contact
  values:
    displayName: Marie Durand
    email: marie.durand@example.org
  context: production
```

Listing 16.2: Instance identity in logiCells YAML syntax

16.3 Relation Identity and this

Some relations are simple structural links. Others are facts that must be tracked. When a relation occurrence has its own history, status, provenance or audit trail, it should have its own identity. In H-Logic-oriented explanations, this is often the role of `this`: it refers to the identity of the relation or fact itself.

For example, the fact that a scheduled item was executed is not merely a link between an item and an action. It may have a timestamp, a result, an error message, an operator and a retry count. The relation occurrence becomes a runtime fact.

```yaml
class:
  name: ScheduledExecutionFact
  classId: lgc#scheduled_execution_fact
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: lgc#scheduled_execution
        facets:
          - main:
              type: hypergraph
              fields:
                - Item: lgc#schedule_action_item
                - Action: lgc#action
                - Queue: lgc#queue
                - ExecutedAt: DateTime
                - Status: String
          - ScheduledExecutionFact:
              type: database
              fields:
                - IdItem:
                    type: String
                    rolePath: Item.Id
                - IdAction:
                    type: String
                    rolePath: Action.Id
                - IdQueue:
                    type: String
                    rolePath: Queue.Id
                - ExecutedAt: DateTime
                - Status: String
```

Listing 16.3: Reified execution fact in YAML model notation

16.4 Entity Identity and me

The notion of `me` is useful when a declaration is read from the point of view of the carried entity or the principal support of a structure. While `this` identifies the current fact or relation occurrence, `me` can designate the entity being described or projected.

This distinction avoids ambiguity in declarations that combine entity properties and relation properties. For instance, a contact assignment relation may have its own status, while the contact itself has a name and email. The assignment and the contact are not the same identity.

16.5 Context

A context is a dimension of validity and interpretation. A declaration may be valid in a production context, a test context, a customer context, a project context, or a security context. Context is not merely a tag. It can affect visibility, constraints, available actions and interpretation.

The current MODEL grammar does not define a generic top-level `context:` object. Contextual semantics are represented explicitly in the model/rule layer. In H-Logic, a context is an ordinary typed role:

```
define .:#Context(me)[label::#string];
define .:#ScheduledIn(item:#ScheduleActionItem, context:#Context);
#production:#Context;
```

Listing 16.4: Context represented as an explicit role

A context such as `projectBeta` should therefore be modeled as a concept instance or contextual interface and passed through relations explicitly, not encoded with an undocumented YAML wrapper or an ambiguous inline function call such as `context(projectBeta)`.

16.6 Conceptual Scope

Scope defines where a declaration applies. A constraint may apply to all scheduled action items, only to scheduled action items in a production context, or only to a specific package. A view may be global, role-specific, or context-specific. An action may be available for all instances of a concept, or only for instances satisfying a particular condition.

Explicit scope prevents accidental generalization. A rule that is correct for structured export actions may not be correct for all export actions. A workflow that is correct for onboarding may not be correct for periodic background jobs.

16.7 Redefinition and Extension

logiCells models often evolve. New facets, constraints and actions may be added over time. The safest way to evolve a model is to distinguish redefinition from extension. Extension adds new meaning while preserving the identity of the existing concept. Redefinition changes the meaning of the concept and must therefore be treated carefully.

Chapter 17

Constraints, Defaults and Validity

A declarative conceptual model is not only a vocabulary. It also records constraints and value policies that the runtime can interpret. This chapter deliberately uses only forms present in the current executable MODEL grammar. Business invariants that are not represented by a MODEL attribute should be expressed by the rule/action layer rather than by inventing new YAML keys.

17.1 Concept Constraints

Under a concept, the current MODEL grammar accepts a `constraints` collection. Its registered constraint forms include `implies`, `equivalent`, `and`, `or`, `xor`, `not` and the generic `constraint`. The named logical forms can refer to another concept with `relatedTo` or carry an expression; the compact YAML display form exposes the related target directly.

```yaml
class:
  name: ScheduleActionItem
  classId: lgc#schedule_action_item
  className: CustomClassItem
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
```

Listing 17.1: Canonical implication constraint

When a constraint is expression-based, the grammar also carries the expression language and an optional polyadic map. The exact expression semantics belong to the corresponding language contract; do not infer a new YAML mini-language from examples.

17.2 Participant Typing and Role Paths

For a relation-like concept, participant expectations can be expressed by typed fields in the conceptual facet and connected to persistence fields through `rolePath`. Nullability belongs to the field contract and is distinct from the logical meaning of the relation.

```yaml
class:
  name: ScheduledExecution
  classId: lgc#scheduled_execution
  className: CustomClassItem
  type: Relation
  concepts:
    - Relation:
        constraints:
          - implies: lgc#relation
        facets:
```

```
          - main:
              type: hypergraph
              fields:
                - Item: lgc#schedule_action_item
                - Action: lgc#action
                - Queue: lgc#queue
                - Context: lgc#execution_context
                - Result:
                    type: lgc#action_result
                    nullable: true
          - ScheduledExecution:
              type: database
              fields:
                - IdItem:
                    type: String
                    rolePath: Item.Id
                    nullable: false
                - IdAction:
                    type: String
                    rolePath: Action.Id
                    nullable: false
                - IdQueue:
                    type: String
                    rolePath: Queue.Id
                    nullable: false
                - IdContext:
                    type: String
                    rolePath: Context.Id
                    nullable: false
                - IdResult:
                    type: String
                    rolePath: Result.Id
                    nullable: true
```

Listing 17.2: Typed participants and persistence role paths

This declaration makes the representation contract inspectable. It does not, by itself, prove every business invariant about a scheduled execution. Cross-object and temporal invariants should be validated by the appropriate H-Logic, action or process contract.

17.3 Defaults

The current field grammar uses `defaultValue`. It is a value policy for a missing field value; it should not be confused with a proof that the value is valid in every business context.

```
concepts:
  - Entity:
      facets:
        - main:
            type: hypergraph
            fields:
              - Priority:
                  type: Integer
                  defaultValue: 0
              - Executed:
                  type: Boolean
                  defaultValue: false
```

Listing 17.3: Field defaults in canonical MODEL YAML

Action parameters also support `defaultValue`, but the action parameter grammar is different from the field grammar.

17.4 Nullability

The MODEL field grammar exposes `nullable`. Use it when absence is part of the field contract.

```
fields:
  - Action:
      type: lgc#action
      nullable: false
  - Result:
      type: lgc#action_result
      nullable: true
```

Listing 17.4: Nullable and non-null fields

Do not substitute undocumented keys such as `required` or `validWhen` in MODEL YAML. If a validity condition depends on execution state, context, permissions or other objects, express that condition in the rule/action layer that owns the invariant.

17.5 Action Parameter Contracts

A direct action parameter is declared under `params`. In the current grammar, every parameter requires `dataType` and `paramType`; `defaultValue` is optional. Direction and return semantics are therefore explicit in the parameter contract rather than inferred from a generic `required` flag.

```
actions:
  - action:
      name: ExecuteScheduledAction
      type: ObjectItem
      params:
        - Context:
            dataType: String
            paramType: ptInput
        - Retry:
            dataType: Boolean
            paramType: ptInput
            defaultValue: false
        - Result:
            dataType: Boolean
            paramType: ptReturn
```

Listing 17.5: Canonical action parameter contract

The accepted `paramType` values are runtime contract values. The declarative grammar defines the shape; the runtime/action implementation defines the operational meaning of each value.

17.6 Fixed Business Invariants

This guide does not document a generic top-level `fixedValues` MODEL construct because it is not part of the current registered MODEL grammar. A business rule such as "this action family is always export" must therefore be represented by an actual supported mechanism: a specialized concept, a declared field/default where appropriate, a constraint expression, H-Logic, or action/process validation.

This distinction is intentional. Documentation examples must not make an unsupported YAML key look canonical merely because the idea is useful.

17.7 Validity Is Layered

Validation occurs at several layers and those layers should not be conflated:

Declarative validity The YAML source obeys the registered grammar: legal children, required attributes and valid structural forms.

Model validity Identifiers, role paths, references and model-specific constraints are coherent enough for the model to be resolved.

Semantic validity H-Logic propositions and queries satisfy the semantic contract that applies to the operation.

Operational validity An action or process is allowed in the current state, context and permission boundary.

A successful syntax validation proves only the first layer unless the invoked tool explicitly performs additional checks. This is why the guide distinguishes MODEL, APPLICATION and SEMANTIC authority throughout the tooling chapters.

Chapter 18

Conceptual Graphs and Hypergraph Memory

logiCells is built around the idea that business meaning should not be scattered between code, tables, routes, screens and documentation. The developer first describes that meaning in MODEL sources. When those declarations or H-Logic sources are compiled semantically, the runtime can materialize them in conceptual hypergraph memory, where concepts, relations, roles and constraints become directly queryable and composable.

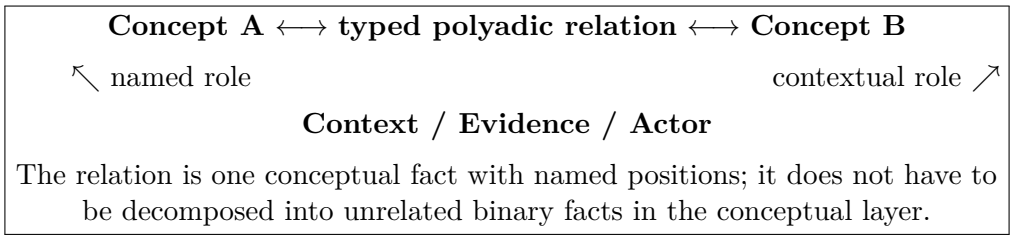


Figure 18.1: Conceptual hypergraph intuition: concepts and contexts participate in typed relations through named roles.

A classical graph links one node to another node. That is useful for many navigation problems, but it is often too weak to express business structures directly. Business facts are rarely only binary. A scheduled action may involve an action path, a target object, a triggering rule, a state, a user, a queue and an execution context. A workflow activity may depend on a base object, a current state, a set of transitions and a publication surface. A conceptual hypergraph allows these structures to be represented as typed, role-bearing relations instead of being flattened too early into tables or object references.

18.1 Why Hypergraph Memory Matters

The role of hypergraph memory is not to replace source files, application metadata, persistence or execution mechanisms. It is the SEMANTIC runtime representation used when conceptual structures must be matched, related, inferred or queried. The same project can therefore be viewed at three different levels: MODEL source, APPLICATION metadata and SEMANTIC hypergraph.

This distinction matters because the word *hypergraph* appears in two places. In a YAML model, a facet whose type is `hypergraph` is a declaration that describes a conceptual projection. Editing that facet is still a MODEL operation on a file. The physical semantic hypergraph is a compiled runtime structure and is not the authority for source editing.

Once compiled, conceptual relations can be explored through indexes, navigation, patterns

and logical operations. The hypergraph is therefore not a passive copy of the source; it is the structural memory used by the semantic runtime while the source file remains the authority for what was declared in MODEL.

18.2 Concepts, Relations and Hyperedges

A concept identifies a meaningful type or structure in the model. It may represent an entity, a relation, a workflow activity, an action, a domain object or a technical capability. A relation connects several conceptual positions. When the relation has more than two positions, it is naturally polyadic. Instead of forcing such a relation through intermediate tables or artificial objects, the hypergraph can keep it as a single typed structure with named roles.

This is especially important for declarative programming. A declaration should say what exists, how it is constrained and how it participates in the model. It should not prematurely decide whether the implementation will be a class field, a database column, a join table, a service parameter or a UI field. Hypergraph memory gives logiCells a neutral conceptual substrate before those projections are selected.

18.3 A YAML Example: Hypergraph and Database Facets

The following simplified example follows the structure used by the actual logiCells YAML model files. The class exposes a conceptual hypergraph facet and a database projection facet. The hypergraph facet keeps the conceptual roles; the database facet describes persistent fields.

```yaml
class:
  classId: lgc#activity
  name: Activity
  className: CustomClassItem
  currentGeneratorName: Common.CustomActivity
  type: Entity

  object:
    className: ObjectItemActivity
    objectName: Activity

  concepts:
    - Entity:
        constraints:
          - implies: CustomActivity
        facets:
          - main:
              type: hypergraph
              fields:
                - ActionPath: String
                - BaseObject: Object
          - Activity:
              type: database
              fields:
                - ActionPath:
                    size: 255
                    readOnly: false
                - BaseObject:
                    readOnly: true
                - FormalNameActivity:
                    size: 100
                    caption: '#FormalNameActivity'
                    rolePath: Activity.FormalName
                    nullable: true
```

Listing 18.1: YAML class with hypergraph and database facets

This example illustrates a central logiCells rule: the same model element can carry several complementary meanings. The `main` facet describes the conceptual structure. The `Activity`

facet describes a database projection. The field `FormalNameActivity` is not just a raw column; it is connected to a conceptual path through `rolePath`. This is how logiCells keeps a bridge between conceptual meaning and operational data.

18.4 Hypergraph Memory and Runtime Execution

Hypergraph memory becomes operational when the runtime uses the declared structure to instantiate objects, resolve paths, evaluate relations, select providers, publish views or execute actions. The important point is that runtime execution does not start from an isolated class or table. It starts from a structured model.

When a field is declared inside a hypergraph facet, it belongs to the conceptual shape of the object. When a field is declared inside a database facet, it belongs to a persistence projection. When a role path is present, the runtime can connect a projected value back to a conceptual relation or property path. This allows logiCells to keep model-level meaning while still supporting practical engineering concerns such as databases, forms, services and data grids.

18.5 Relation with AI and Semantic Memory

The conceptual hypergraph is also the natural place to connect symbolic structures with AI-oriented representations. The API documentation discusses embeddings as annotations or tensorial projections of hypergraph nodes, and distinguishes conceptual storage from computational projection. This distinction is essential: a vector embedding may help search, classify or compare nodes, but the conceptual hypergraph remains the structured memory that explains what the nodes mean.

For AI assistants, this is a strong architectural advantage. An assistant can operate on named concepts, typed roles, contextual relations and declared actions instead of receiving only raw text or database rows. The hypergraph gives the assistant a structured operating surface.

18.6 Persistent Structural Pattern Evaluation

H-Logic structural queries use persistent hypergraph indexes. The runtime chooses a selective physical pivot, enumerates real relations, validates fixed and wildcard slots, and only then applies any active type-lattice mapping. Physical index selection is separate from logical alternatives such as exact type, subtype and supertype.

This separation is particularly important for inherited instances. A type alternative carries a mapping from real candidate slots to the apparent slots requested by the query. The relation itself remains the indexed candidate; a descendant type node is not used as a replacement for the physical relation pivot.

Legacy transient storage support is isolated behind the compile-time symbol `TransientLegacy`. The normal execution path and the declarative semantics described in this guide use the persistent hypergraph.

Chapter 19

Conceptual Layers, Projections and Views

A logiCells model is not a single flat description. It is a layered structure. The same concept may be seen as a conceptual object, a persistent record, a collection row, a view item, an action target, a workflow node or an AI-accessible memory element. Declarative conceptual programming requires a clear distinction between the model itself and its projections.

This chapter introduces the role of layers, facets, projections and views. The goal is to prevent a common architectural mistake: confusing the business concept with one of its representations.

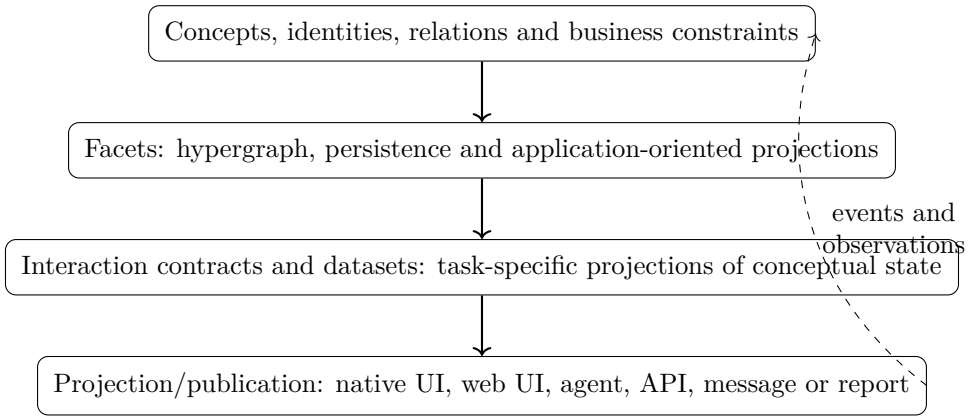


Figure 19.1: Publication is a projection of conceptual meaning, not the place where business meaning originates.

19.1 The Concept Is Not the Projection

A concept is the stable semantic element. A projection is a way to expose that element for a specific purpose. For example, an `Activity` concept may be projected as a hypergraph structure, a database record, a workflow state, a UI list row, a service payload or an assistant-readable item. These projections are useful, but none of them alone defines the concept.

This is why logiCells models distinguish facets. A facet is a named projection surface attached to a concept. The `main` facet is commonly used for the hypergraph structure. A database-named facet may describe persistence. A view or collection layer may then reuse the declared structure to expose the concept to users or services.

19.2 Facets as Projection Boundaries

A facet is a boundary between meanings. It tells the runtime how to interpret a set of fields in a specific projection context. In a hypergraph facet, fields describe conceptual roles. In a database facet, fields describe persistent columns or projected values. In a UI-oriented view, fields may describe displayable items, captions, action buttons or navigation links.

The following YAML fragment shows this separation. The same class contains a conceptual layer and a database layer.

```yaml
class:
  classId: lgc#activity
  name: Activity
  type: Entity

  concepts:
    - Entity:
        facets:
          - main:
              type: hypergraph
              fields:
                - ActionPath: String
                - BaseObject: Object
          - Activity:
              type: database
              fields:
                - ActionPath:
                    size: 255
                    readOnly: false
                - BaseObject:
                    readOnly: true
                - FormalNameActivity:
                    rolePath: Activity.FormalName
                    nullable: true
```

Listing 19.1: Facet separation in YAML

The important point is not the syntax itself, but the modeling discipline. The `main` facet expresses what the object means conceptually. The database facet expresses how part of that meaning is made persistent or queryable through a storage-oriented surface.

19.3 Role Paths as Cross-Layer Bridges

A `rolePath` connects a projected field to a conceptual path. This is a powerful mechanism because it prevents projected fields from becoming semantically isolated. A displayed or stored value can still be traced back to the conceptual structure that gives it meaning.

For example, a field such as `FormalNameActivity` may be projected in a database facet, while its `rolePath` points to `Activity.FormalName`. The field can be stored, displayed or queried, but it remains linked to the conceptual role it represents.

```yaml
FormalNameActivity:
  size: 100
  caption: '#FormalNameActivity'
  rolePath: Activity.FormalName
  nullable: true
```

Listing 19.2: A projected field connected to a conceptual role path

In documentation and modeling reviews, role paths should be treated as semantic commitments. They are not only technical paths; they explain why a field exists and where its meaning comes from.

19.4 Interaction Contracts as Model Projections

The modern VIEW layer is better understood as an interaction-contract layer. A contract does not define the concept and does not prescribe one screen. It describes how a concept or a collection of concepts participates in an interaction: what information is available, what information may need to be supplied, which actions can be invoked, which events can be observed and which functional interaction components are relevant.

The participant can be a human user or an autonomous agent. A native renderer may materialize the contract as VCL/FMX controls, a web renderer may materialize it as HTML or reactive components, and an agentic runtime may use it as an interaction plan: present useful information, ask only for missing values and invoke the same declared application capabilities without displaying a form.

This principle is essential for portability. The same contract can be enacted through several channels while the conceptual model and action semantics remain stable. The historical term *view* remains in APIs and file types; `meta/views/contracts` makes the channel-independent role explicit.

19.5 Layered Evolution

Layers also make model evolution safer. A concept can gain a new database field without changing its conceptual identity. A view can be redesigned without changing the underlying relation structure. A service can expose a subset of fields without redefining the domain. Conversely, a conceptual change can be propagated to several projections when it is intentional.

This is one of the practical benefits of logiCells. It separates stable domain meaning from publication and persistence decisions, while keeping them connected through metadata.

Part IV

H-Logic and Declarative Expressions

Chapter 20

H-Logic Essentials

H-Logic is the compact logical language of the logiCells hypergraph layer. It complements the native YAML model representations: YAML declare application artifacts, while H-Logic expresses typed conceptual structures, facts, logical relations, queries, projections and executable commands directly in the vocabulary of the graph.

The current grammar is designed around explicitness. Roles are named, contexts are ordinary roles, named-term references use the `&` prefix, interface attachment uses `::`, and queries expose their result through `return`. These choices make H-Logic suitable both for human reasoning and for machine-generated conceptual operations.

20.1 Why H-Logic Matters

H-Logic gives logiCells a language between static metadata and general-purpose code. It can express a conceptual rule without hiding it in a script, formulate a query without reducing the model to SQL, and describe an interface attachment without changing the identity of the supporting object.

This has several practical benefits:

- **Business meaning stays visible.** Concepts, roles and relations remain present in the expression.
- **Polyadic facts remain first-class.** A fact can keep all of its participants instead of being decomposed prematurely into binary links.
- **Queries and commands share one vocabulary.** The same conceptual forms can be asserted, queried or used as the body of another definition.
- **Interfaces can be attached without replacing identity.** The `::` operator keeps the support object distinct from the attached conceptual view.
- **The language is testable.** Public examples can be aligned with parser and evaluation tests rather than treated as informal pseudo-code.

H-Logic is therefore valuable for rules, semantic validation, conceptual search, model projections, explainable automation and AI agents that need to operate on explicit structures instead of opaque implementation details.

It also supports an earlier lifecycle role: H-Logic can act as a verified specification pivot between requirements and generated declarative models. This workflow is developed in Chapter 29, “From Business Specifications to Validated H-Logic” and applied to a temporal stock-management case study in Chapter 30, “Case Study: Temporal ERP Stock Analysis in H-Logic”.

20.2 Relationship with YAML

The three forms have different responsibilities.

```yaml
class:
  classId: lgc#activity
  name: Activity
  type: Entity
  concepts:
    - Entity:
        facets:
          - main:
              type: hypergraph
              fields:
                - ActionPath: String
                - BaseObject: Object
```

Listing 20.1: YAML model declaration

```
define .:#Activity(me)[actionPath::#string, baseObject:#Object];
```

Listing 20.2: H-Logic conceptual definition

The YAML fragments declare model metadata. The H-Logic fragment exposes the conceptual shape: an `Activity` has a distinguished support role `me`, a runtime string value boxed by `#string`, and a conceptual object reference.

20.3 Terms, Types and Structural Use

A typed term uses the form `x:#T`. A named conceptual instance uses a formal symbol such as `#alice:#Person`. An anonymous instance uses `_:#Person`.

```
define .:#Person(me)[name::#string];
#alice:#Person;
_:#Person;
```

Listing 20.3: Typed terms and instances

The parser permits some forward type annotations, but structural use requires the target type to have been defined. For example, attaching an interface or supplying arguments to a concept must refer to a known structural definition.

20.4 Roles and Attributes

Roles appear in parentheses. Attributes appear in brackets. This distinction mirrors the conceptual distinction between structural positions and descriptive properties.

```
define .:#worksFor(person:#Person, company:#Company, context:#Context);
define .:#Order(me)[amount::#float, status:#OrderStatus];
```

Listing 20.4: Roles and attributes

A role may carry a structural order annotation:

```
define .:#SecondOrderFact(item:#Item, context^2:#Context);
define .:#ThirdOrderEvidence(evidence^3:#Evidence, context^2:#Context);
```

Listing 20.5: Role-order metadata

The $\hat{\mathtt{n}}$ annotation belongs to the role metadata of the declared type. It is not written on quantified variables.

20.5 Applying a Known Type with Local Role Names

H-Logic can apply a type that is already known by the semantic graph while giving the occurrence local role names. This is especially useful for expressions and constraints.

```
.:#add(Amount, Tax)[result_1]
.:#power(Amount, 2)[squared_amount]
```

Listing 20.6: Local role names over a known operator type

If `#add` or `#power` is already defined by the active semantic environment, these expressions do not redefine the type. The existing type remains authoritative for its structure and arity; `Amount`, `Tax`, `result_1` and `squared_amount` name the roles of this particular pattern.

A local role name is not a hidden `#variable` concept. Likewise, the literal `2` constrains one position of the `#power` pattern but does not become a public role. This keeps the H-Logic surface readable while allowing the parser to build the internal role mappings required by the composite.

20.6 Interface Attachment and Boxing

The operator `::` attaches a conceptual interface to an existing support. It preserves the identity of the support while creating a conceptual box or view.

```
define .:#Auditable(me)[auditLevel::#integer, lastAuditDate::#Date];
doc42::#Auditable;
```

Listing 20.7: Interface attachment

The compact form desugars conceptually to an anonymous interface instance:

```
doc42::#Auditable;
_:#Auditable(doc42);
```

Listing 20.8: Compact and explicit forms

When the attached box must itself be named and reused, use the complete named form:

```
auditBox:#Auditable(doc42)[auditLevel=2];
```

Listing 20.9: Named interface instance

Runtime primitive values are also exposed conceptually through boxing:

```
define .:#PrimitiveDocument(me)[title::#string, checksum::#string];
doc42:#PrimitiveDocument[title='HDR chapter', checksum='abc123'];
```

Listing 20.10: Primitive value boxing

20.7 Named Terms

An explicit named-term reference uses `&name`. The prefix is semantic and is not part of the logical name.

```
projectBeta := #projectBeta;
exists target=&projectBeta on .:#VisibleIn(label42, target);
```

Listing 20.11: Explicit named-term binding

A named-term reference cannot be used as a role name. The following declaration is invalid:

```
define .:#B(&x:#A);
```

Listing 20.12: Invalid role name

Use a role name and constrain its value instead:

```
define .:#BFixed(x:#A=&a);
define .:#BDefault(x:#A default &a);
```

Listing 20.13: Role with fixed or default named term

20.8 Quantifiers

H-Logic supports universal, existential and aggregation quantifiers. The most common modeling forms are `any`, `every`, `exists` and `some`.

```
exists y:#B on .:#R(y);
every y:#B on .:#R(y);
exists target=&projectBeta on .:#VisibleIn(label42, target);
any x:#A, exists y:#B on .:#Pair(x, y);
```

Listing 20.14: Typed and value-bound quantifiers

A quantified variable may have a type restriction or an equality binding, but not both in the same compact declaration. Therefore `any y:#A=&a` is not a valid form.

20.9 Logical Relations and Complements

Binary logical operators include implication, equivalence, derivation and exclusion. Complementary properties are attached with `_[...]`.

```
#A ->_[weight=1, context=#c] #B;
#Diagnosis :-_[weight=0.8, context=#clinical] #SymptomPattern;
#Candidate !-_[context=#validation, reason=#missing_required_role] #Bottom;
#A <->_[context=#shared_context, confidence=0.95 as conf] #B;
```

Listing 20.15: Logical relations with complementary properties

These complements keep provenance, confidence, context or rule metadata attached to the logical relation itself.

20.10 Commands and Queries

A bare assertion is a tell command. The explicit forms `tell`, `do`, `ask` and `?` make the command intention visible.

```
projectBeta:#Context;
tell projectBeta:#Context;
do projectBeta:#Context;
ask .:#A(x) and .:#R(x, y) return x;
```

Listing 20.16: Tell, do and ask commands

Queries no longer require a prefixed variable list such as `?x`. Free roles in the target expression form the query environment, and the `return` expression selects the result. The query envelope also accepts the reasoning modes `homomorphic`, `unifying` and `resolving`, plus scope options for instances, types, subtypes, supertypes and masked values. The complete execution and result contract is described in Chapter 25, “H-Logic Query Modes, Scope and Result Semantics”.

```
? .:#Person(Tom)[name='Tom Hanks']
  and .:#like(Tom, Anne)
return (Tom, Anne);
```

Listing 20.17: Query returning conceptual results

20.11 Composite and Projected Definitions

A concept can be defined as a logical composition of existing structures.

```
define .:#ContextualRule(rule, context) as
  .:#Rule(rule)
  and .:#Context(context)
  and .:#validIn(rule, context);
```

Listing 20.18: Composite definition

The public role list may project a richer internal body, provided every public role is constrained by that body. This allows a stable public conceptual interface over implementation-rich compositions.

```
define .:#PersistedU(me, x);
define .:#PersistedV(me_, x, y);
define .:#ProjectedF(x, y) as
  .:#PersistedU(me, x)
  and .:#PersistedV(me_, x, y);
```

Listing 20.19: Projected composite definition

20.12 Where H-Logic Adds the Most Value

H-Logic is especially useful when the application must preserve and execute explicit meaning:

- semantic constraints and derivations;
- conceptual queries over graph structures;
- explainable rule evaluation with context and confidence;
- projected concepts built from several source structures;
- interface attachment to existing objects or primitive values;
- AI agents that need typed, queryable and governable actions over conceptual memory.

The language does not replace YAML, and it does not replace functionalP. YAML are MODEL sources, H-Logic expresses and exchanges SEMANTIC knowledge, and functionalP supplies procedural behavior or expression source where appropriate. An APPLICATION may compile both model and semantic artifacts, but the three authorities remain distinct.

20.13 H-Logic as a Canonical Exchange Form

The runtime can also serialize supported conceptual values to canonical H-Logic and parse H-Logic back through the official grammar. For an application developer, this matters because the same language used for rules and queries can also provide a compact, inspectable view of accumulated knowledge. It avoids inventing a separate representation for diagnostics, assistants or generated semantic summaries.

The practical boundary is:

`logiCells knowledge` → `canonical H-Logic serialization` → `review, query or assistant context`

When H-Logic comes back from an external source or an LLM, the direction is deliberately stricter:

`candidate H-Logic` → `official parser` → `symbolic validation` → `transaction or operationalizable intention`

A generated text is therefore a *proposal*, not active knowledge. The developer does not need to manage parser internals or serializer classes; the important rule is that canonical H-Logic is produced by the platform and candidate H-Logic is accepted only after the engine has parsed and validated it.

Chapter 21

Declaring Concepts and Instances

A logiCells model starts by declaring a class-level artifact, then by describing the conceptual structure that the runtime will materialize. In the YAML model syntax, the root element of a model file is usually `class`. The declarative structure contract defines this root with attributes such as `classId`, `name`, `className`, `currentGeneratorName` and `type`. The same structure can be represented in XML, but YAML is the compact syntax used throughout this book for application model declarations.

A concept declaration is not only a label. It tells the runtime what kind of model artifact is being described, how it should be identified, what runtime object class can support its instances, and which conceptual facets belong to it. An instance is then an occurrence of this declared structure: a folder, an activity, a device, a domain object, a scheduled action item, or any other object governed by the same metadata discipline.

21.1 The Class Declaration

The following fragment is adapted from the `Activity` model. It shows the minimal upper structure of a logiCells model class.

```yaml
class:
  classId: lgc#activity
  name: Activity
  className: CustomClassItem
  currentGeneratorName: Common.CustomActivity
  type: Entity
  object:
    className: ObjectItemActivity
    objectName: Activity
```

Listing 21.1: YAML class declaration

The declaration introduces a model-level class identifier, a human-readable model name, the runtime class item used to manage the model, and the object representation used for instances. The important point for this book is not the implementation class itself, but the separation between the conceptual model and the runtime structures that can materialize it.

21.2 Concepts inside a Class

Inside a class declaration, the `concepts` section defines the conceptual interpretation of the class. The grammar accepts a `concept` element under `class/concepts`; in YAML examples, this may appear as a named map entry such as `Entity`. logiCells uses this concept section to associate constraints and facets with the declared class.

```yaml
class:
```

```yaml
  classId: lgc#activity
  name: Activity
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: CustomActivity
        facets:
          - main:
              type: hypergraph
              fields:
                - ActionPath: String
                - BaseObject: Object
```

Listing 21.2: YAML concept declaration under a class

The concept `Entity` indicates how the class participates in the conceptual layer. The constraint expresses that an `Activity` implies or specializes another conceptual class, here `CustomActivity`. The `main` facet, typed as `hypergraph`, gives the conceptual roles or slots carried by the model.

21.3 Entity Class Type and Entity Concept

The class attribute `type: Entity` and the conceptual `Entity` record serve different purposes. The class type classifies the model declaration. It does not, by itself, create the semantic `Entity` concept used by conceptual registration and semantic constraints.

When an application entity must participate in the semantic layer, declare the `Entity` concept explicitly.

```yaml
class:
  classId: app#work_item
  name: WorkItem
  className: CustomClassItem
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
        facets:
          - main:
              type: hypergraph
```

Listing 21.3: Entity class with an explicit semantic Entity concept

A useful diagnostic rule follows directly from this distinction: if runtime logs report that a class has no `Entity` record, inspect the `concepts` declaration before changing its procedural implementation.

21.4 Implicit Me Identity for Entities

In H-Logic, a non-singleton entity is propositionally represented with its implicit instance identity, conventionally written `me`. This identity is part of the semantic representation; it is not an application field that must be added to the YAML model.

```
define .:#WorkItem(me);
```

Listing 21.4: Implicit entity identity in H-Logic

The corresponding declarative model may have an empty main hypergraph facet. Do not add a field named `me` or `Me` merely to reproduce the H-Logic notation. Business identity fields and persistence identifiers are separate projections.

21.5 Implication Is a Semantic Mapping

An `implies` constraint is not merely an object-oriented inheritance annotation. The source and target concepts must be semantically compatible so that the runtime can construct the required role mapping.

For ordinary non-singleton entities, the implicit `Me` identity normally provides the common positive role:

```
UnitTest(Me)     -> Command(Me)
AgentProcess(Me) -> Process(Me)
```

Listing 21.5: Conceptual implication over entity identity

If runtime diagnostics report that no polyadic map can be constructed between two concepts, first verify that both semantic concepts are actually declared and that their positive roles are compatible. Do not add arbitrary fields only to silence the mapping error.

21.6 Conceptual Fields

Fields inside a hypergraph facet are not merely database columns. They describe conceptual roles carried by the instance. In the `Folder` model, the main facet contains roles such as `Name`, `Description`, `ParentFolder`, `LastChanged` and `Model`. These names reflect the current generation conventions, but their semantic function is clear: they define what the conceptual instance can carry.

```yaml
class:
  classId: lgc#folder
  name: Folder
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
        facets:
          - main:
              type: hypergraph
              fields:
                - Name: String
                - Description: String
                - ParentFolder: lgc#folder
                - LastChanged: DateTime
                - Model: Boolean
```

Listing 21.6: YAML hypergraph facet for a folder concept

The role `ParentFolder` is especially important. It declares that a folder can carry a reference to another folder. This is already a conceptual relation at the model level, even before one decides how it should be serialized into a table, exposed in a view, or navigated in a user interface.

21.7 Instances as Runtime Occurrences

The model does not itself create a concrete folder or activity. It declares what a folder or activity is. A runtime instance is then an occurrence of the declared class. For example, a concrete folder instance may have a name, a description, a parent folder and a model flag. These values conform to the roles declared in the conceptual facet.

The distinction is essential:

- the class declaration defines the conceptual type;

- the object declaration indicates the runtime representation for instances;
- the concept declaration gives the semantic interpretation;
- the facets describe how the same concept appears in different projections;
- runtime instances carry values according to these declarations.

21.8 Database Facets and Instances

The database facet shows how a conceptual instance can be projected into persistent fields. It should not be confused with the concept itself. A folder is not reducible to the `lgcFolder` table. The table is one projection of the conceptual model.

```yaml
facets:
  - Folder:
      type: database
      map: lgcFolder
      fields:
        - IdFolder:
            generatorName: Common.Folder
            size: 32
            caption: '#Identifier'
            rolePath: Id
        - ClassIdFolder:
            size: 32
            defaultValue: lgc#folder
            caption: ClassId
            rolePath: ClassId
        - Name:
            fieldName: FolderName
            size: 100
            caption: '#FolderName'
            defaultValue: '#NewFolder'
        - IdParentFolder:
            size: 32
            packLinkedObject: true
            caption: '#ParentFolder'
            rolePath: ParentFolder.Id
            nullable: true
```

Listing 21.7: YAML database facet for folder persistence

This declaration maps conceptual identity and role paths to persistent fields. The field `IdParentFolder` is a storage projection of the conceptual role `ParentFolder.Id`. The model keeps the conceptual path explicit, instead of hiding it behind an arbitrary foreign-key column.

21.9 H-Logic Reading

The same model can be read conceptually with H-Logic. This is not a replacement for the YAML model file. It is a compact logical reading of the conceptual structure.

```
define .:#Folder(me, parentFolder:#Folder)
  [name::#string, description::#string,
   lastChanged::#Date, model::#boolean];
```

Listing 21.8: H-Logic reading of a folder concept

The H-Logic view emphasizes the conceptual type and its roles. The YAML view gives the complete application model, including runtime class, object representation, facets, persistence projection, captions, default values and role paths.

Chapter 22

Declaring Relations and Reified Facts

logiCells models relations in two complementary ways. Some relations appear as roles inside a concept, such as the `ParentFolder` role of a folder. Other relations are declared as model classes in their own right, such as `BelongsTo`. This second form is essential when a relation must carry its own identity, metadata, persistence projection, actions, visibility rules or lifecycle.

This chapter explains the difference between a role reference and a reified relation. A role reference connects one concept to another inside the structure of an entity. A reified relation is itself an entity-like model whose purpose is to represent a fact between two or more participants.

22.1 Role References

The `Folder` model contains a role pointing to another folder. In the hypergraph facet, this appears as a conceptual field.

```yaml
facets:
  - main:
      type: hypergraph
      fields:
        - Name: String
        - ParentFolder: lgc#folder
```

Listing 22.1: YAML role reference in a hypergraph facet

In the database facet, this conceptual role is projected through role paths.

```yaml
facets:
  - Folder:
      type: database
      map: lgcFolder
      fields:
        - ClassIdParentFolder:
            size: 32
            packLinkedObject: true
            caption: '#ParentFolderClass'
            rolePath: ParentFolder.ClassId
            nullable: true
            defaultValue: lgc#folder
        - IdParentFolder:
            size: 32
            packLinkedObject: true
            caption: '#ParentFolder'
            rolePath: ParentFolder.Id
            nullable: true
```

Listing 22.2: YAML role paths for a linked parent folder

The field names are storage-level names. The `rolePath` attributes preserve the conceptual path. This lets the runtime know that these fields are not isolated strings, but projections of the linked object `ParentFolder`.

22.2 Reified Relations

A reified relation is a relation represented as its own model. The `BelongsTo` model illustrates this pattern. It declares a conceptual entity with two participants, `EntityA` and `EntityB`. The relation itself receives a class identifier, a runtime object representation and persistence fields.

```yaml
class:
  classId: lgc#belongs_to
  name: BelongsTo
  className: CustomClassItem
  currentGeneratorName: Common.BelongsTo
  type: Entity
  object:
    className: CustomObjectItem
    objectName: BelongsTo
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
        facets:
          - main:
              type: hypergraph
              fields:
                - EntityA: lgc#entity
                - EntityB: lgc#entity
```

Listing 22.3: YAML declaration of a reified relation model

Although the model uses `type: Entity`, its conceptual intention is relational. This is not a contradiction. A reified relation is a fact that can itself be identified, stored, filtered, displayed, secured and extended.

22.3 Why Reify a Relation?

A simple role is enough when the relation is merely a structural reference. A reified relation becomes useful when the fact must be manipulated as an object of the model. This happens when the relation has its own lifecycle, when it must be listed in a collection, when actions must be attached to it, when audit information is required, or when it carries additional roles.

For example, `BelongsTo(EntityA, EntityB)` can represent membership, containment, grouping or ownership. Because the relation is reified, one can later add context, validity period, source, confidence, state or workflow information without breaking the conceptual model.

22.4 Persistence of Reified Relations

The database facet of `BelongsTo` projects the relation and its participants into persistent fields. The relation has its own identity fields, and each participant is projected through role paths.

```yaml
facets:
  - Agency:
      type: database
      map: lbsAgency
      fields:
        - IdBelongsTo:
            generatorName: Common.Agency
            size: 32
            caption: '#Identifier'
            rolePath: Id
```

```yaml
        - ClassIdBelongsTo:
            size: 32
            defaultValue: lgc#belongs_to
            caption: ClassId
            rolePath: ClassId
        - IdEntityA:
            size: 32
            caption: '#IdEntityA'
            rolePath: EntityA.Id
            nullable: true
        - ClassIdEntityA:
            size: 32
            caption: '#ClassIdEntityA'
            defaultValue: lgc#entity
            rolePath: EntityA.ClassId
            nullable: true
        - IdEntityB:
            size: 32
            caption: '#IdEntityB'
            rolePath: EntityB.Id
            nullable: true
        - ClassIdEntityB:
            size: 32
            caption: '#ClassIdEntityB'
            defaultValue: lgc#entity
            rolePath: EntityB.ClassId
            nullable: true
```

Listing 22.4: YAML database projection of a reified relation

The pattern is important: the relation has its own identity, and each role has its own projected identity. This is richer than a classical foreign key because the role path keeps the semantic relationship explicit.

22.5 H-Logic Reading

In H-Logic, the conceptual meaning of `BelongsTo` can be expressed as a relation with two typed roles.

```
define .:#BelongsTo(entityA:#Entity, entityB:#Entity);
```

Listing 22.5: H-Logic reading of a reified BelongsTo relation

If the relation itself must be addressed, one can give it an identity through a named fact rather than using an anonymous occurrence.

```
define .:#BelongsTo(entityA:#Entity, entityB:#Entity);
fact1:#BelongsTo(entityA, entityB);
```

Listing 22.6: Named relational fact

The named fact is addressable independently from its participants. This allows the relation itself to carry attributes, provenance, confidence, validity or workflow state.

22.6 Role Paths as Conceptual Bridges

Role paths are one of the most important bridges between conceptual modeling and persistence. A field such as `IdEntityA` is not just a database column. Its `rolePath: EntityA.Id` tells the runtime that it is the identifier of the participant playing the `EntityA` role in the relation.

This gives the model a stable semantics across projections. The same relation can be read as a hypergraph fact, stored as a row, displayed in a view, exposed through a service or navigated by a tool.

Chapter 23

Contexts, Interfaces and Named Terms

Contexts, interfaces and named terms solve three different problems: where a fact is valid, how an existing support is exposed conceptually, and how a known logical term is referenced explicitly. The current H-Logic grammar keeps these concerns separate.

23.1 Contexts Are Ordinary Roles

H-Logic no longer relies on a special context suffix. A context is modeled as an ordinary typed role.

```
1 define .:#Context(me)[label::#string];
2 define .:#VisibleIn(item:#Item, context:#Context);
```

Listing 23.1: Context as a role

This makes context compositional. A project, tenant, workflow execution, security scope or temporal frame can be represented by a normal concept and passed through relations explicitly.

23.2 Role Order

When a context participates at a higher structural order, the role declaration can carry `n̂` metadata.

```
1 define .:#SecondOrderFact(item:#Item, context^2:#Context);
2 define .:#ThirdOrderEvidence(evidence^3:#Evidence, context^2:#Context);
```

Listing 23.2: Higher-order context roles

The order annotation belongs to the declared role. Quantified variables continue to use plain variable names.

23.3 Interfaces Preserve Support Identity

An interface concept is commonly defined with a distinguished `me` role. The `::` operator attaches that interface to an existing support.

```
1 define .:#Auditable(me)[auditLevel::#integer];
2 doc42::#Auditable;
```

Listing 23.3: Defining and attaching an interface

The support `doc42` remains the support object. The attachment introduces a conceptual view over it. This is useful when the same object must participate in several conceptual interfaces without being duplicated or retyped destructively.

23.4 Direct Instance versus Interface Attachment

The operators `:` and `::` have different meanings.

```
#alice:#Person;
#alice::#Customer;
```

Listing 23.4: Direct instance and attached interface

The first statement declares `#alice` as a direct instance of `#Person`. The second attaches the `#Customer(me)` interface to the existing support `#alice`.

23.5 Explicit Named Terms

A reference such as `&projectBeta` resolves a named term from the environment.

```
projectBeta := #projectBeta;
exists target=&projectBeta on .:#VisibleIn(label42, target);
```

Listing 23.5: Explicit named-term resolution

The prefix disambiguates a value reference from a role declaration. It is particularly useful in generated rules and parameterized conceptual queries.

23.6 Fixed and Default Values

A role can be fixed to a named term or given a default value.

```
define .:#FixedScope(context:#Context=&globalContext);
define .:#DefaultScope(context:#Context default &globalContext);
```

Listing 23.6: Fixed and default role values

A fixed value constrains resolution. A default supplies a value when no explicit value is provided. They should not be confused.

23.7 Complex Role Domains

A role type can be a named type or a parenthesized logical type expression.

```
define .:#Choice(role:(#A or #B));
```

Listing 23.7: Logical role domain

For reusable subset semantics, define a named type in the graph and use that name as the role domain.

```
.:#SubsetContext -> .:#Context;
define .:#ContextFamily(context:#SubsetContext);
```

Listing 23.8: Named subset type

Constructor-like role domains such as `Subset(#Context)` are not the stabilized form.

23.8 Why These Distinctions Matter

Explicit contexts improve traceability. Interface attachment enables multiple conceptual views over the same support. Named terms make value bindings unambiguous. Together, these mechanisms make H-Logic suitable for explainable rules, contextual authorization, semantic projections and agent operations over conceptual memory.

Chapter 24

Queries over Conceptual Structures

H-Logic queries operate on conceptual structures rather than on storage tables. They expose roles, relations, contexts and logical composition directly, which makes the query intention readable and traceable.

24.1 Ask and Question-Mark Forms

Both `ask` and `?` introduce a query. The expression after `return` defines the result.

```
ask .:#A(x) and .:#R(x, y) return x;
```

Listing 24.1: Simple conceptual query

```
? .:#Person(Tom)[name='Tom Hanks']
  and .:#like(Tom, Anne)
return (Tom, Anne);
```

Listing 24.2: Query returning several terms

Free roles in the query expression form the environment. Older forms with prefixed query variables such as `?x := ...` are not part of the current grammar. A query may additionally select a reasoning mode and scope options. Homomorphic matching is the default; unifying mode combines compatible operands; resolving mode proves goals through facts and rules. See Chapter 25, "H-Logic Query Modes, Scope and Result Semantics" for the full semantics.

24.2 Typed Quantification

Quantifiers constrain the roles traversed by a conceptual pattern.

```
exists y:#B on .:#R(y);
every y:#B on .:#R(y);
any x:#A, exists y:#B on .:#Pair(x, y);
```

Listing 24.3: Existential and universal constraints

`exists` requires at least one matching value. `every` constrains all values associated through the target relation. `any` establishes a source domain for the following pattern.

24.3 Binding to a Named Term

A quantified role can be fixed to an explicit named term.

```
projectBeta := #projectBeta;
exists target=&projectBeta on .:#VisibleIn(label42, target);
```

Listing 24.4: Query binding to an environment value

This is useful for parameterized queries generated from a selected project, tenant, workflow execution or security context.

24.4 Logical Composition

Queries can compose conceptual patterns with `and`, `or`, arithmetic expressions and relational operators.

```
? .:#Person(tom, #Yesterday)[name='Tom Hanks']
  and r:#like(tom, Anne, #Yesterday)
  and s:#hate(Anne, John, #Yesterday)
return (tom, r, s);
```

Listing 24.5: Composed conceptual query

Naming relation facts such as `r` and `s` allows the query to return or constrain the facts themselves, not only their participants.

24.5 Query as an Executable Bridge

A query may return a command.

```
ask .:#A(me, name, context) and (name='toto')
return do _:#love(me, #you, context);
```

Listing 24.6: Query followed by a conceptual action

This form is valuable for declarative automation: the condition and the resulting conceptual operation remain in the same language and can be inspected, traced and governed.

24.6 Why H-Logic Queries Are Valuable

Compared with storage-oriented queries, H-Logic keeps role names and conceptual relations visible. This supports explainable search, semantic validation, model-aware assistants and rules that can survive changes in physical persistence. Runtime providers may still translate or optimize execution, but the source expression remains anchored in the conceptual model. Result cardinality is represented polymorphically: no answer is `nil`, one answer is returned directly, and several answers form a linked conceptual list.

Chapter 25

H-Logic Query Modes, Scope and Result Semantics

The `ask` command is a common declarative envelope for several reasoning operations. A query always provides a conceptual target and a return projection, while its *mode* selects the reasoning operation and its *options* select the part of the type and instance space that may participate.

This separation is important. The query expression describes *what* is sought. The mode describes *how* compatible structures are found. The options describe *where* the search is allowed to range.

25.1 The Three Query Modes

The current runtime recognizes three modes.

`homomorphic` Finds persisted conceptual structures that match a structural pattern. This is the default mode.

`unifying` Unifies two or more compatible conceptual operands and returns the materialized canonical result.

`resolving` Resolves a goal through facts and Horn-style rules, including recursive and tabled resolution where supported by the program.

A mode changes the reasoning algorithm, not the vocabulary of the query. Concepts, roles, named facts and return projections keep the same conceptual meaning.

```
1 ask homomorphic .:#Person(p)[name='Ada']
2   return p;
3
4 ask unifying (.:#Candidate(a), .:#Candidate(b))
5   return (a, b);
6
7 ask resolving .:#ancestor(alice, descendant)
8   return descendant;
```

Listing 25.1: The same query envelope with explicit reasoning modes

When no mode is written, the query is interpreted as `homomorphic`.

25.2 Homomorphic Pattern Matching

Homomorphic mode searches the conceptual hypergraph for persisted values compatible with the query pattern. A fixed role value is an exact constraint. An omitted or unconstrained role is a wildcard.

```
// Exact structural pattern
ask .:#worksFor(#alice, #logiCells, #production)
  return this;

// Partial pattern: company and context remain unconstrained
ask .:#worksFor(#alice, company, context)
  return (company, context);
```

Listing 25.2: Exact and partial homomorphic patterns

A wildcard is the absence of a constraint. It is not a request to compare the candidate with a distinguished `nil` value. This distinction allows partial patterns to use the same structural iterator as exact patterns without excluding populated candidate slots.

When a query contains several pattern elements, they are evaluated in query order. Their matched values are appended to the result in that same order.

25.3 Unifying Mode

Unifying mode combines compatible operands into one canonical conceptual value. It preserves repeated-variable equalities, recursively unifies nested constructed values and rejects constructor, arity or rigid-value conflicts.

```
ask unifying (
  .:#Pair(x, #right),
  .:#Pair(#left, y)
)
return (x, y);
```

Listing 25.3: Unifying compatible conceptual operands

A unifying query requires at least two operands. With more than two operands, unification proceeds across the whole sequence; an incompatibility in any operand makes the query empty. The result projection can return descriptors, constants or a list assembled from both.

The materialized result is canonical: equivalent requests can share the same conceptual result rather than creating structurally duplicate values.

25.4 Resolving Mode

Resolving mode treats the query as a logical goal. The resolver selects clauses by canonical constructor, applies fresh variables for each rule application, propagates substitutions from left to right through conjunctive bodies and projects only the variables requested by the query.

```
ask resolving .:#reachable(#warehouseA, destination)
  return destination;
```

Listing 25.4: Resolving a goal through facts and rules

Multiple facts or alternative clauses may produce multiple answers. Duplicate proofs are deduplicated. A recursive cycle without a base fact produces no answer; tabled recursive programs can produce transitive answers without repeatedly expanding the same unresolved goal.

25.5 Scope Options

Scope options are orthogonal to the mode. They select whether the requested type itself, its instances, its position in the type lattice or combinations of these may be returned.

`masked` Hides values excluded by the active visibility or masking policy.

instances Includes matching instances in addition to the default scope.

instances only Restricts results to instances.

types Includes matching types in addition to the default scope.

types only Restricts results to types.

subtypes Includes the requested type and its descendants.

subtypes only Restricts the lattice portion to descendants of the requested type.

supertypes Includes the requested type and its ancestors.

supertypes only Restricts the lattice portion to ancestors of the requested type.

The **subtypes** and **supertypes** traversals are reflexive: the requested type remains an eligible exact alternative unless an **only** restriction excludes it through the selected result policy.

```
ask homomorphic subtypes instances .:#Document(item)[title=title]
  return (item, title);
```

Listing 25.5: Homomorphic instance search across descendants

25.6 Direction in the Type Lattice

Inheritance is represented by the logical relation `#supertype(Map, Source, Target, Context)`. `Source` is the subtype and `Target` is the supertype.

For a requested type `T`:

- the internal `>` direction follows relations whose target is `T` and obtains their source, therefore discovering subtypes;
- the internal `<` direction follows relations whose source is `T` and obtains their target, therefore discovering supertypes;
- `=` keeps the exact type;
- `*` leaves the kind unconstrained.

These symbols describe iterator direction and are useful when reading traces or engine diagnostics. Declarative H-Logic authors normally select the same behavior through **subtypes** and **supertypes** options.

25.7 Inherited Instances and Mapped Views

A subtype may reorder, project or identify the roles of its supertype. Therefore a descendant instance cannot always be returned as a raw value when it is queried through an ancestor pattern. The runtime returns a mapped conceptual view that preserves the identity of the real instance while exposing roles in the requested type's apparent order.

```
define .:#Parent(left:#Value, right:#Value);
define .:#Child(first:#Value, second:#Value);

// Conceptual inheritance mapping: Child.first -> Parent.right,
// Child.second -> Parent.left
#Child ->_[map=#[[0,1],[1,0]]] #Parent;
```

Listing 25.6: A subtype whose roles are presented through a parent mapping

A query for `Parent` with `subtypes` may match a persisted `Child(first=A, second=B)`. The returned value still points to that child instance, but its apparent parent slots are `left=B` and `right=A`.

The mapping invariant is:

> real candidate slots map to requested apparent slots.

This invariant is maintained for exact, descendant and ancestor traversal. Upward traversal reverses or composes mappings as needed before exposing the result.

25.8 Polyadic Maps Preserve Equality Information

A polyadic mapping is not merely a permutation. Its normal form can represent projection, repeated positions and equality between fields. For example, several real slots may participate in one equality component, or a public type may omit a private descendant slot.

The normal polyadic form is therefore the semantic source of truth. Runtime access resolves a unique correspondence in the required direction:

- candidate comparison resolves a real slot to its apparent pattern slot;
- mapped-value access resolves an apparent requested slot to its real instance slot.

A correspondence is used only when it is unique. Ambiguous mappings remain visible as equality information rather than being silently reduced to an incorrect simple permutation. Implementations may cache validated correspondences for performance, but the cache is derived from the normal polyadic map and does not replace it.

25.9 Result Contract

Every query return value follows one polymorphic contract:

- zero results produce `nil`;
- one result produces that `PNodeValue` directly;
- several results produce a linked `PListValue`.

Any node value can therefore be interpreted through list semantics. A singleton behaves as a one-element list whose head is the value itself and whose tail is `nil`. Appending a second value promotes the result to a proper linked list.

The return expression may itself be a list. The runtime does not add an artificial outer singleton list around it. This keeps the declarative result shape faithful to the expression written after `return`.

```
// No match: nil
ask .:#Missing(x) return x;

// One match: the value itself
ask .:#Person(p)[name='Ada'] return p;

// Several matches: linked list of values
ask .:#Person(p) return p;

// One projected tuple: the tuple itself, not [[p, name]]
ask .:#Person(p)[name=name] return (p, name);
```

Listing 25.7: Return-shape examples

25.10 Persistent Iteration and Performance

Structural pattern evaluation uses the persistent relation indexes of the hypergraph. The iterator first chooses a physical pivot with a selective index, enumerates real candidate relations, checks the candidate against exact or wildcard constraints, then applies any active type mapping before returning the value.

The physical pivot and the logical type alternatives are separate concerns. A descendant type is a filter alternative with an associated mapping; it is not itself a substitute for the relation-index pivot. This separation keeps exact and inherited searches on the same indexed execution path.

The default query path works against persistent relation indexes. If a legacy integration requires a different storage mode, validate its query semantics and performance separately on the deployed runtime.

25.11 Practical Selection Guide

Use `homomorphic` when the desired answer already exists as a persisted conceptual structure. Use `unifying` when several descriptions must be merged into one compatible canonical value. Use `resolving` when the answer must be proved through facts and rules.

Then add scope options only for the conceptual region that is required. In particular, do not request `subtypes` merely to compensate for an imprecise type declaration: descendant traversal changes both candidate selection and role mapping and should remain an intentional semantic choice.

25.12 H-Logic as the Single Knowledge Matching Authority

Use `KnowledgeQuery.SemanticPattern` to provide the canonical H-Logic query. Other request fields, including `WorldContextSelector`, `EvidenceRequirements` and `PolicyContext`, select scope and apply safeguards; they do not define a second graph query language.

When FunctionalP constraints participate in a knowledge query, the runtime translates their compiled expression into H-Logic before evaluating it. Built-in predicates such as `#add` have established arity and type contracts. A local H-Logic expression does not redefine them.

A Polyadic Map represents both role correspondence and equality constraints. Matching role names can provide an implicit mapping only where the relation structure supports it. For different role names or more complex equivalence classes, provide an explicit mapping. This preserves meaning when projecting relations or traversing inherited types.

Use the `ask` modes supported by the installed runtime for interactive graph queries. Additional query-compilation capabilities, where offered, must be checked independently against the target CLI or API; successful `ask` execution is not evidence that a separate compilation command is available.

Chapter 26

Rules, Constraints and Tested Patterns

Rules and constraints make a model safer than a collection of forms and scripts. They define what must hold for the model to be valid, what may be inferred, and which combinations of roles, states and values are meaningful.

26.1 Constraints as Model Commitments

A constraint is a commitment carried by the model. It may state that a concept implies another concept, that a value is required, that a default value should be applied, or that a role must satisfy a specific type.

```
constraints:
  - implies: lgc#entity
```

Listing 26.1: A concept-level implication

The implication says that the concept participates in the semantic and runtime contract of `lgc#entity`. It is more than inheritance in a class hierarchy. It is a statement about conceptual compatibility.

26.2 Rules as Executable Interpretation

A rule may validate a state, derive a value, authorize an action, route an event, or select a view. In logiCells, rules should be expressed as close as possible to the model element they govern. A validation rule belongs near the concept or action it validates. A routing rule belongs near the event or process it controls.

26.3 Expression Constraints and Role Compatibility

Some constraints use an expression rather than only naming another concept. For the application developer, the important contract is simple: the expression is compiled into a semantic target, then logiCells binds the source concept to that target through compatible conceptual roles.

```
source roles
  + expression roles
  -> compatible role mapping
  -> semantic constraint
```

Listing 26.2: Constraint binding intuition

An explicit mapping, when declared, remains authoritative. Without one, matching relies on compatible roles exposed by both sides. If an expression refers only to roles that do not exist on the source concept, the constraint is invalid for that source and should be reported or ignored rather than guessed positionally.

This rule is useful when reviewing a constraint: role names are part of its conceptual interface. Renaming a role in an expression without a corresponding source role or explicit map can change whether the constraint applies at all.

26.4 Tested Patterns

A tested pattern is a small reusable modeling shape that has been validated across models. Examples include:

- an entity with a hypergraph facet and a database projection;
- a reified relation with role paths for both participants;
- a process object with `Start`, `Validate` and `Finalize` actions;
- a published action with typed parameters;
- a message model with creation date, reception date and sender metadata;
- a view bound to a cursor and a collection path.

Tested patterns are important because they make model evolution safer. A team can add new concepts without reinventing identity, persistence, event handling and view binding each time.

26.5 Constraint Placement

A constraint should be placed at the lowest level where it remains true. A constraint that always applies to a concept belongs to the concept. A constraint that only applies during a process belongs to the process. A constraint that only applies to service exposure belongs to the publication layer.

26.6 Design Rules

Do not hide validity only inside scripts. Use scripts to execute, not to be the only source of semantic truth. Keep constraints visible in the model. Test the common patterns independently from the application workflow. Prefer small constraints with clear scope over large procedural validations.

26.7 Summary

Rules and constraints turn a declarative model into a controlled executable structure. Tested patterns make those structures repeatable and evolvable.

Chapter 27

Common H-Logic Misreadings

H-Logic is compact, but its operators have precise meanings. The following distinctions reflect the current grammar and parser tests.

27.1 A Role Name Is Not a Named-Term Reference

The prefix `&` refers to a named value in the environment. It cannot replace a role name.

```
// Invalid
// define .:#B(&x:#A);
```

Listing 27.1: Invalid and valid forms

The valid form is:

```
define .:#B(x:#A=&a);
```

Listing 27.2: Role constrained to a named term

27.2 Type Restriction and Equality Binding Are Separate

A quantified variable accepts either a type restriction or an equality binding.

```
exists y:#A on .:#R(y);
exists y=&a on .:#R(y);
```

Listing 27.3: Valid quantified forms

The compact combined form `exists y:#A=&a` is not part of the grammar.

27.3 Contexts Are Roles, Not Function Calls

The form `context(projectBeta)` is not the current context syntax. Use a context role and bind it explicitly.

```
define .:#VisibleIn(item:#Item, context:#Context);
projectBeta:#Context;
_:#VisibleIn(label42, projectBeta);
```

Listing 27.4: Explicit contextual relation

27.4 Colon and Double Colon Are Different

`:` declares or types a term. `::` attaches an interface to an existing support.

```
#alice:#Person;
#alice::#Customer;
```

Listing 27.5: Typing versus interface attachment

Treating these forms as interchangeable loses the distinction between an object and a conceptual box attached to it.

27.5 Structural Uses Require Defined Types

A simple forward annotation may be accepted, but structural application requires a known definition.

```
define .:#Context(me);
trace42::#Context;
```

Listing 27.6: Define before structural use

27.6 Query Variables Are Not Prefixed with Question Marks

The current query syntax uses `ask` or `?` before the expression. It does not use identifiers such as `?x`.

```
? .:#A(x) and .:#R(x, y) return x;
```

Listing 27.7: Current query form

27.7 Role Order Is Declaration Metadata

The annotation `context^2` belongs to a role declaration. Quantified variables use the plain role name.

```
define .:#VisibleIn(item:#Item, context^2:#Context);
any context:#Context, exists item on .:#VisibleIn(item, context);
```

Listing 27.8: Declared role order and plain quantified role

27.8 Composite Definitions Must Constrain Public Roles

A projected definition may hide internal roles, but every public role must be constrained by the body. Adding a public role that never appears in the body is invalid because it would expose an unconstrained interface.

27.9 A Wildcard Is Not a Nil Constraint

An omitted or unconstrained role in a homomorphic pattern means that any candidate value is accepted at that position. It does not mean that the candidate role must contain `nil`. Treating a wildcard as a nil-valued filter incorrectly rejects populated relations.

27.10 Subtype Search Does Not Return Raw Descendants Unchanged

A subtype may present its roles in a different order or identify several roles through a polyadic map. When a descendant instance is returned through a parent query, the runtime may return a mapped conceptual view. The underlying identity remains the descendant, while the visible slot order follows the requested parent type.

27.11 A Polyadic Map Is Not Only a Permutation

A normal polyadic map preserves projection, repeated positions and equality components. Reducing it unconditionally to a simple permutation loses information and can reverse the apparent-to-real access direction. A fast positional cache is valid only when it has been derived from a unique correspondence in the normal map.

27.12 Local Role Names Do Not Redefine a Known Type

When a semantic environment already knows a type, a pattern may give that occurrence local role names without changing the type definition.

```
1 .:#add(Amount, Tax)[result_1]
```

Listing 27.9: Known type with local role names

Here `Amount`, `Tax` and `result_1` are role names for this occurrence. They are not new structural metadata for `#add` and they are not `#variable` nodes.

27.13 A Literal Is Not a Public Role

In `.:#power(Amount, 2)[result_1]`, the literal `2` constrains the operand position. It is not exposed as a role that another concept can map to.

27.14 No Shared Role Can Mean No Valid Mapping

An inferred polyadic map must correspond to compatible roles. If a source concept and a constraint expression share no role that can be aligned, the constraint is not applicable to that source. Do not invent a positional map or a synthetic role to force it to fit.

27.15 Nested Logical Groups Must Remain Groups

The expressions `A and (B or C)` and `A or (B and C)` are not equivalent to a flat list of three operands. The parser must preserve the nested sub-composite. Canonical serialization may flatten repeated instances of the same operator, but different operators must retain the grouping required by their meaning.

27.16 One Result Is Not Necessarily a One-Element Container

The query result contract is polymorphic. A single result is returned directly; it is not wrapped in an additional list. Several results form a linked list, and no result is `nil`. A return expression that already constructs a tuple or list keeps that shape without an extra outer wrapper.

27.17 H-Logic Does Not Replace YAML, VIEW XML or functionalP

YAML are declarative MODEL sources. H-Logic expresses SEMANTIC knowledge and can also be compiled from or toward application structures. functionalP provides procedural behavior and can serve as the source language for expressions whose semantic form is compiled separately. Keeping these authorities distinct prevents a source editor from pretending to know the assembled application and prevents procedural code from becoming the only source of semantic truth.

Part V

From Business Specifications to Applications

Chapter 28

Two Paths from Business Specifications to logiCells Applications

logiCells supports two complementary ways to turn a business specification into an executable application. Both start from the language of the domain, both favor concepts over implementation objects, and both converge on declarative logiCells models. They differ mainly in *when* the project introduces H-Logic and what it wants to validate first.

The two paths are:

1. **Model-first**: business text → declarative models → application, with optional H-Logic generation to obtain a compact semantic view of the accumulated knowledge and to check its coherence.
2. **H-Logic-first**: business text → H-Logic → symbolic coherence checks in the logiCells engine → generated model skeletons → operationalization code attached to actions associated with ontology verbs.

Model-first loop Business text → declarative models → application → runtime feedback → optional H-Logic synthesis → coherence review → refined models
H-Logic-first loop Business text → H-Logic candidate → symbolic validation → model skeletons → operationalizations → application → new knowledge → refined H-Logic

Figure 28.1: The two development paths are feedback loops and may feed each other.

Neither path is universally better. The right choice depends on the uncertainty of the domain, the amount of existing knowledge, the importance of formal consistency, and how quickly the project needs a working application surface.

The paths are also *cyclical*. They describe two useful entry points and two dominant feedback loops, not two irreversible pipelines. A project may iterate several times inside one loop, cross into the other loop when its risks change, and later return. The stable element is the conceptual knowledge carried by the platform, not the order in which artifacts are first produced.

28.1 Two Entry Points, Two Feedback Loops

A model-first project usually learns from the running application. The model is refined from user feedback, runtime observations and newly discovered business distinctions. H-Logic can then summarize the knowledge accumulated across those iterations and expose semantic gaps that are hard to see from individual model files.

`business text` → `declarative models` → `application` → `runtime and business feedback` → `H-Logic synthesis when useful` → `coherence review` → `refined models`

An H-Logic-first project usually learns from semantic validation. The knowledge specification is refined until its vocabulary, bindings, constraints and deductions are coherent enough to generate application structures. Once those structures run, operational feedback may reveal new knowledge and start another H-Logic cycle.

`business text` → `H-Logic` → `symbolic validation` → `model skeletons` → `operationalizations and application` → `runtime and business feedback` → `enriched H-Logic`

In both cases, the team alternates between *expressing meaning*, *making that meaning executable*, and *learning from execution*. This is a core logiCells development pattern.

28.2 Path A: Model-First, H-Logic as a Semantic Synthesis

Use the model-first path when the application structure is already reasonably understood and the primary goal is to produce working business software with minimal code.

`business specification` → `conceptual model` → `declarative YAML` → `views, actions, workflows and services` → `working application` → `optional H-Logic synthesis`

In this path, the developer starts by identifying business concepts, relations, states, fields, actions and projections. The declarative model is the main construction surface. The platform generates or executes as much infrastructure as possible from these declarations.

H-Logic can then be generated from the accumulated conceptual knowledge when the team needs a more compact semantic representation. This synthesis is useful for:

- reviewing the vocabulary and relations accumulated across several models;
- checking whether concepts introduced by different features remain coherent;
- making implicit conceptual dependencies visible;
- querying the knowledge at a higher semantic level;
- giving a human reviewer or an assistant a concise representation of the application's meaning;
- preparing later symbolic or neuro-symbolic capabilities without forcing them into the first iteration.

The important point is that H-Logic is *optional in the construction loop*. A team can build a useful logiCells application declaratively before it needs advanced logical analysis.

28.2.1 Typical contexts for the model-first path

Prefer this path when:

- the business domain is familiar and its main entities and workflows are already agreed;
- the project must deliver screens, services or workflows quickly;
- the first concern is replacing imperative CRUD and glue code with declarative models;
- the ontology is expected to emerge progressively from application development;
- formal inference is useful, but not required to decide the initial model structure;
- the team has strong business expertise but limited expertise in logic programming.

A typical departmental application, workflow application, operational dashboard or CRUD-heavy business system can often start here.

28.3 Path B: H-Logic-First, Models as Operationalization

Use the H-Logic-first path when semantic ambiguity and consistency are the main risks of the project.

`business specification` → `H-Logic specification` → `logiCells coherence checks` → `validated conceptual knowledge` → `generated model skeletons` → `operationalization of actions` → `working application`

The project first translates the business specification into H-Logic. The logiCells parser and symbolic runtime are then used to check the structure, bindings, relations, constraints and deductions before application artifacts are generated.

Once the semantic specification is sufficiently stable, it can seed the declarative model layer. The generator creates skeletons for concepts, relations, fields, actions and other artifacts that can be derived safely. Application-specific decisions such as persistence, publication, views and interaction remain explicit generation or design choices.

28.3.1 Verbs and operationalization

A useful distinction is to keep *what the business means* separate from *how the runtime performs it*.

Business verbs in the ontology express intentions such as `Validate`, `Compute`, `Reserve`, `Approve`, `Publish` or `Notify`. These intentions can be projected as logiCells actions. The generated action declaration remains conceptual; implementation code is added only where an operationalization is actually required.

`ontology verb` → `declared action` → `candidate operationalization` → `minimal implementation code`

This keeps custom code at the edge of the conceptual model. The code implements a business intention; it does not become the place where the business vocabulary and rules are hidden.

28.3.2 Typical contexts for the H-Logic-first path

Prefer this path when:

- the domain contains many interacting rules, constraints or derived facts;
- several teams or sources use partially overlapping vocabularies;
- contradictions, temporal scope or context must be identified before implementation;
- the application is based on knowledge accumulated from specifications, documents or heterogeneous systems;
- symbolic inference, explanation or semantic query is a central project requirement;
- an LLM or another generative system participates in knowledge acquisition and its proposals must be validated before becoming executable knowledge;
- generated application models should be a projection of an already validated semantic specification.

Knowledge-intensive applications, decision support, regulation-heavy domains, semantic integration and neuro-symbolic systems are natural candidates for this path.

28.4 How the Two Paths Converge

The two paths do not create two platform architectures. They converge on the same logiCells runtime.

Model-first	**H-Logic-first**
Text clarifies the business feature.	Text clarifies the business knowledge.
Conceptual YAML is produced early.	H-Logic is produced early.
Application feedback refines the model.	Symbolic validation refines the semantics.
H-Logic can synthesize accumulated knowledge later.	Declarative model skeletons are projected after validation.
Code is added only for missing operational behavior.	Code is added to operationalize declared actions/verbs.

At convergence, both paths use the same principles:

- concepts and relations carry business meaning;
- declarative metadata describes application structure;
- actions express intentions before implementations;
- views and services are projections of the conceptual model;
- H-Logic is the structured symbolic representation when logical analysis is needed;
- custom code is reserved for operationalizations that cannot be obtained declaratively;
- symbolic validation remains authoritative for logical validity;
- neural or LLM capabilities may propose, rank, classify or assist, but do not bypass the symbolic and transactional boundaries of the runtime.

28.5 Choosing a Path

The following table gives a practical default. It is a project-start decision aid, not a permanent architectural constraint: a project can begin model-first and introduce H-Logic later, or begin H-Logic-first and then spend most of its life evolving declarative models.

Project context	**Recommended starting path**	**Reason**
Well-known business process, short delivery cycle	Model-first	Get a working conceptual application quickly; add semantic synthesis when the model grows.
Existing CRUD or workflow application to modernize	Model-first	Replace implementation structure progressively with concepts, actions and projections.
Emerging ontology built during product discovery	Model-first, then H-Logic synthesis	Let the application reveal useful concepts, then consolidate the vocabulary and coherence.
Rule-heavy or regulated domain	H-Logic-first	Validate vocabulary, constraints and deductions before generating operational artifacts.
Knowledge integration from heterogeneous sources	H-Logic-first	Resolve semantic overlap and contradictions before deciding application projections.
LLM-assisted knowledge acquisition	H-Logic-first	Treat generated knowledge as candidate H-Logic, parse and validate it before activation or operationalization.
UI-centric prototype with simple semantics	Model-first	Avoid introducing logical complexity before it creates value.
Decision support requiring explanations and inference	H-Logic-first	The symbolic model is part of the product, not only a later diagnostic view.

28.6 Switching Loops Is Normal

Many projects deliberately move between the two loops. A team may build its first modules model-first, generate an H-Logic synthesis when the conceptual surface becomes large, detect inconsistencies, and then use the stabilized H-Logic knowledge to generate the next family of model skeletons. Conversely, a project can begin H-Logic-first to stabilize a regulated or knowledge-intensive core, then evolve most later features directly in declarative models once the semantic foundation is trustworthy.

A mixed cycle can therefore look like:

```
text → models → application feedback → H-Logic synthesis → coherence review →
        new or regenerated model skeletons → application feedback
```

The trigger for switching loops should be pragmatic:

- switch toward H-Logic when semantic ambiguity, contradictions, rule interaction or cross-model coherence become the dominant risk;

- switch toward model-first iteration when the knowledge is sufficiently stable and delivery of views, services, workflows or operational behavior becomes the dominant need;
- revisit H-Logic whenever accumulated application knowledge deserves a new semantic synthesis.

This is not a failure to choose a methodology. It is one of the benefits of using a common conceptual runtime: application structure and symbolic knowledge can progressively reinforce each other without changing platform or abandoning previous work.

28.7 Keep Expertise Proportional to the Project

A business developer should not need to understand parser internals, unification algorithms, ranking implementations or neural backends to choose either path.

The guide therefore uses the following progression:

- start with business vocabulary and intentions;
- use declarative models for application structure;
- introduce H-Logic when semantic synthesis or validation creates concrete value;
- activate symbolic functions through tested patterns rather than engine internals;
- activate neural functions through platform capabilities and safe defaults;
- expose implementation details only when the developer intentionally crosses an extension boundary.

The goal is not to turn every application developer into a logic or machine-learning specialist. The goal is to let business developers obtain more semantic, symbolic and neural capabilities while keeping most of their work at the conceptual level.

28.8 Using Skills to Enter Either Loop

AI Skills are especially useful at the entry points of these cycles. They can turn an informal specification into a structured business analysis, recommend the loop that currently provides the best feedback, and then apply the corresponding generation discipline.

In a model-first cycle, a Skill can propose concepts, declarative YAML, views, actions and services, then later generate an H-Logic synthesis for coherence review. In an H-Logic-first cycle, a Skill can translate the specification into H-Logic, prepare it for validation by the official parser and symbolic runtime, and then generate declarative model skeletons plus explicit operationalization points for ontology verbs.

The developer should not need to know the internals of unification, ranking, serialization or neural backends to use these workflows. Skills expose safe, project-oriented operations; the executable declarative grammar and H-Logic engine remain the validation authorities. A team can also ask the assistant to switch loops when the dominant project risk changes rather than restarting the project with a different methodology.

28.9 Where to Continue

For the model-first path, continue with the application-building chapters and the declarative grammar reference, especially Chapter 38, “Design Decision Guide: What to Create and When”. When the accumulated model needs a semantic synthesis, continue with the H-Logic chapters in Part IV, “H-Logic and Declarative Expressions”.

For the H-Logic-first path, continue with Chapter 29, “From Business Specifications to Validated H-Logic”, then Chapter 31, “From Validated H-Logic to Declarative Models”. The stock-management case study illustrates the complete semantic-analysis path.

Chapter 29

From Business Specifications to Validated H-Logic

This chapter develops the *H-Logic-first* path introduced in Chapter 28, "Two Paths from Business Specifications to logiCells Applications". H-Logic can be used before application modeling as a controlled pivot between a business specification and the executable YAML artifacts of a logiCells application. In this path, it is not merely a rule language added after the model has been designed. It is the intermediate representation in which meaning is made explicit, reviewed, completed and verified before it is operationalized. Projects that primarily need rapid application construction can instead begin model-first and generate H-Logic later as a semantic synthesis.

The development chain becomes:

`business specification` → `H-Logic analysis` → `human review` → `LLM-assisted inference` → `LGC validation` → `generated declarative models`

This workflow reduces the risk of generating structurally valid metadata from an ambiguous or incomplete requirement. The YAML model is generated only after the semantic decisions have become visible and testable.

This is not necessarily a one-time front-end phase. After generated models are operationalized and executed, business feedback can produce new H-Logic statements, new constraints or revised vocabulary. The project can therefore return to semantic analysis, validate the new knowledge, regenerate only the affected skeletons and continue the application loop. In the opposite direction, a model-first project can enter this workflow later by generating an H-Logic synthesis from accumulated models.

29.1 Why Introduce a Logical Pivot?

Natural-language requirements are easy to write and discuss, but they routinely leave important questions unanswered:

- Which nouns represent identified entities, and which represent relations or values?
- Which vocabularies are open, and which are complete enumerations?
- Which conclusions depend on time, context or provenance?
- Which rules are definitions, constraints, deductions or operational policies?
- Which derived facts must be materialized for frequent queries?

- Which statements were confirmed by a domain expert, and which were inferred by an assistant?

Generating a `.model.yaml` file directly from prose hides these decisions inside the generated artifact. H-Logic exposes them in a representation that remains close to the domain vocabulary while being suitable for parsing, typing, propagation and testing.

29.2 The Three Representations

The workflow uses three representations with different responsibilities.

29.2.1 Natural-language specification

The specification expresses intent in the language of domain experts. It is the source for human validation, but it may contain synonyms, omissions, implicit assumptions and statements whose scope is unclear.

29.2.2 H-Logic semantic specification

H-Logic makes the domain structure explicit:

- types and identified entities;
- roles, attributes and polyadic relations;
- complete vocabularies and instances;
- temporal and contextual arguments;
- implications, diagnostics and query projections;
- facts expected from inference.

This representation is suitable for human review, LLM-assisted analysis and engine validation.

29.2.3 logiCells declarative artifacts

YAML operationalize the validated semantics as runtime artifacts:

- classes, concepts and facets;
- fields and role paths;
- persistence projections;
- manifests and packages;
- views, actions, events, services and workflows.

The transition from H-Logic to YAML is not a purely textual translation. It adds runtime decisions such as persistence, publication, view design and execution lifecycle.

29.3 The Three Validation Loops

A reliable analysis process separates human review, LLM-assisted inference and engine verification.

29.3.1 Human semantic review

The human reviewer remains responsible for business meaning. The reviewer confirms:

- the vocabulary and its definitions;
- what must have a durable identity;
- the participants and roles of each relation;
- the time and context carried by facts;
- invariants, exceptions and business consequences;
- the object to which a conclusion actually applies.

A frequent modeling error is to attach a conclusion to the evidence rather than to the business object concerned by the conclusion. For example, a stock position can provide evidence that a product must be reordered, but the recommendation concerns the product, warehouse and time.

29.3.2 LLM-assisted inference under symbolic rules

An LLM can accelerate the analysis by proposing:

- candidate concepts and relations;
- missing type declarations;
- temporal roles that were implicit in the prose;
- consistency rules and diagnostic facts;
- query patterns and useful indexed projections;
- clarification questions for the reviewer;
- candidate mappings to logiCells models.

The LLM is constrained by the existing ontology, the H-Logic grammar, accepted modeling conventions and engine feedback. It must not be treated as the authority for executable semantics. Its result is a proposal that must remain attributable and verifiable.

29.3.3 Syntax and semantic verification by the LGC engine

The engine supplies two distinct validation levels.

Syntax validation checks that the source is accepted by the grammar: definitions are well formed, roles and attributes use supported syntax, implications are parseable, and commands use valid forms.

Semantic and execution validation checks the resolved meaning and behavior: types exist, arguments are compatible, variables are bound, temporal values unify correctly, propagation produces expected facts, indexed projections remain coherent, and retraction removes conclusions when their last justification disappears.

Passing the parser does not prove that the ontology behaves correctly. Parser acceptance and logical execution must be reported separately.

29.4 Status and Provenance of Analysis Statements

A specification workflow should distinguish source statements from assistant proposals and engine-validated results. A simple analysis ledger can use three statuses:

confirmed accepted by a human reviewer or an authoritative source;

inferred proposed by an LLM or derived during analysis, but not yet accepted as executable meaning;

validated accepted by the relevant parser or semantic engine test.

The ledger itself can be represented declaratively:

```yaml
analysis:
  - id: STOCK-RULE-003
    statement: Reserved stock cannot exceed physical stock.
    status: confirmed
    source: business-specification

  - id: STOCK-QUERY-004
    statement: Invalid positions should be searchable by warehouse and time.
    status: inferred
    proposedBy: llm-analysis
    basedOn:
      - STOCK-RULE-003
      - operational-diagnosis-needs

  - id: STOCK-PROJECTION-004
    statement: InvalidStockPositionByWarehouseTimeProduct is valid.
    status: validated
    validator: lgc-hlogic-engine
    tests:
      - invalid-position-propagation
      - temporal-index-preservation
```

Listing 29.1: Analysis ledger for requirements and deductions

The YAML ledger does not replace the H-Logic ontology. It records why statements exist, who introduced them and which validation phase accepted them.

29.5 A Repeatable Construction Method

The following method can be reused for ERP, workflow, risk, scheduling and other application domains.

29.5.1 Step 1: Write a compact specification

Start with a small text that names the business objects, observations, policies, conclusions and invariants. Avoid implementation terms at this stage.

29.5.2 Step 2: Extract and normalize the vocabulary

Create a vocabulary table. Record synonyms and choose one canonical name per concept. Ask whether each term denotes an entity, relation, value, event, state, policy or conclusion.

29.5.3 Step 3: Decide where identity is required

Use `me` only when a concept must be an identified object.

```
define .:#StockMovement(me)[
  product:#Product,
  warehouse:#Warehouse,
  quantity:#Quantity,
  occurredAt:#Time
];

define .:#ReorderRequired(
  product:#Product,
  warehouse:#Warehouse,
  at:#Time
);
```

Listing 29.2: Identified entity and pure derived relation

An identified movement can later receive document references, provenance, corrections and audit state. The derived reorder fact needs participants and context, but no independent identity.

29.5.4 Step 4: Declare foundational types before structural use

Declare the types that constrain roles and attributes before building richer definitions.

```
define .:#Product(me);
define .:#Warehouse(me);
define .:#Time(me);
define .:#Quantity(me);
```

Listing 29.3: Foundational domain types

This prevents an assistant or generator from silently treating domain concepts as untyped values.

29.5.5 Step 5: Identify complete business vocabularies

A finite, explicitly closed vocabulary should be represented as a complete type and direct instances.

```
define .:#MovementKind(me);
#MovementKind::#Complete;

#Receipt:#MovementKind;
#Reservation:#MovementKind;
#Release:#MovementKind;
#Shipment:#MovementKind;
```

Listing 29.4: Complete movement-kind vocabulary

The distinction is important: `::` attaches the higher-order complete property, while `:` declares direct instances.

29.5.6 Step 6: Propagate time and context through the whole rule

If a conclusion depends on a temporal state, the same time variable must remain present from the source facts to the conclusion.

```
.:#StockPosition(position)[
  product=product,
  warehouse=warehouse,
  at=t,
  available=available
]
and
.:#ReorderPolicy(policy)[
  product=product,
```

```
  warehouse=warehouse,
  validAt=t,
  threshold=threshold
]
and
.:#LessThan(available, threshold)
->
.:#ReorderRequired(product, warehouse, t);
```

Listing 29.5: Temporal propagation through a reorder rule

This prevents a past stock position from being combined accidentally with a future policy.

29.5.7 Step 7: Express calculations through typed support relations

Arithmetic or external computations can be exposed as typed support relations.

```
define .:#Subtract(
  left:#Quantity,
  right:#Quantity,
  result:#Quantity
);

define .:#LessThan(
  left:#Quantity,
  right:#Quantity
);
```

Listing 29.6: Typed arithmetic support relations

The ontology states the conceptual dependency. The engine or a registered capability supplies the actual calculation.

29.5.8 Step 8: Derive positive diagnostic facts

Represent invalidity as an explicit fact rather than only as the absence of validity.

```
define .:#InvalidStockPosition(
  position:#StockPosition,
  product:#Product,
  warehouse:#Warehouse,
  at:#Time
);
```

Listing 29.7: Explicit diagnostic fact

A positive diagnostic fact is queryable, explainable, indexable and usable by views, services, agents and workflows.

29.5.9 Step 9: Design query access from real usage

Native hypergraph indexes should be used when they already support the access pattern. Introduce `#Indexed` projections only for attribute-to-slot materialization, alternative leading-role orders or frequently queried derived conclusions.

```
define .:#ReorderRequiredByWarehouseTimeProduct(
  warehouse:#Warehouse,
  at:#Time,
  product:#Product
);

#ReorderRequiredByWarehouseTimeProduct::#Indexed;

.:#ReorderRequired(product, warehouse, at)
->
.:#ReorderRequiredByWarehouseTimeProduct(
```

```
  warehouse,
  at,
  product
);
```

Listing 29.8: Indexed projection for a query-oriented order

An indexed projection must not invent new business meaning. It materializes an existing fact in an access-oriented order.

29.5.10 Step 10: Define expected deductions before generation

Create small fact sets with explicit expected results. These examples serve simultaneously as review material, semantic tests and generation acceptance criteria.

29.6 The Role of Questions in LLM-Assisted Analysis

A useful assistant does not only generate declarations. It identifies decisions that require human confirmation. Typical questions include:

- Does this state need a durable identity, or is it a derived relation?
- Is this vocabulary complete, or can projects extend it?
- Does this rule apply at the event time, the processing time or an effective-policy time?
- Should an invalid state prevent persistence, or should it remain observable as a diagnostic?
- Which query orders are frequent enough to justify materialization?
- Is the conclusion a property of the evidence object or of another business object?

The unanswered questions are deliverables of the analysis. They must not be silently resolved by model generation.

29.7 Validation Gates

The transition to application generation should occur only after explicit gates.

Gate 1 — Vocabulary review Domain terms and identities are accepted.

Gate 2 — Grammar acceptance The ontology is accepted by the current H-Logic parser.

Gate 3 — Semantic resolution Types, roles and references resolve correctly.

Gate 4 — Propagation tests Representative facts produce the expected deductions and diagnostics.

Gate 5 — Retraction and justification tests Derived facts remain supported and disappear correctly when their last justification is removed.

Gate 6 — Generation readiness Runtime decisions not contained in the ontology are recorded for the model-generation phase.

The next chapter applies this method to temporal ERP stock management. Chapter 31, “From Validated H-Logic to Declarative Models” then shows how validated meaning becomes logiCells YAML models and application artifacts.

Chapter 30

Case Study: Temporal ERP Stock Analysis in H-Logic

This case study transforms a small stock-management specification into an H-Logic ontology suitable for human review, LLM-assisted completion, parser validation, logical propagation and later generation of logiCells models.

The current ontology used for this example has been accepted by the H-Logic parser. Parser acceptance validates grammar, not the complete behavior of inference, indexing, justification and retraction. Those aspects require engine tests.

30.1 Business Specification

The module manages products stored in warehouses. It records four movement kinds: receipt, reservation, release and shipment. For each product and warehouse, the system must determine the stock state at a given time:

- physical quantity on hand;
- reserved quantity;
- available quantity.

A reorder policy defines a threshold and target level for a product and warehouse at a given time. The business rules are:

- available stock equals on-hand stock minus reserved stock;
- physical and reserved quantities cannot be negative;
- reserved stock cannot exceed physical stock;
- replenishment is required when available stock is below the applicable threshold;
- all deductions retain the time at which their source facts are valid.

30.2 Review the Vocabulary Before Writing Rules

The first review separates identified business objects from pure relations.

Term	Proposed form	Reason
Product	identified entity	Durable business identity and reference target.
Warehouse	identified entity	Shared participant of movements, positions and policies.
StockMovement	identified entity	May carry audit, document and correction history.
StockPosition	identified temporal state	Must be found, explained and potentially corrected.
ReorderPolicy	identified temporal policy	Has an effective time and may evolve independently.
ReorderRequired	derived relation	A conclusion connecting product, warehouse and time.
MovementKind	complete vocabulary	The current specification names an exhaustive set.

30.3 Foundational Types and Complete Vocabulary

```
define .:#Product(me);
define .:#Warehouse(me);
define .:#Time(me);
define .:#Quantity(me);

define .:#MovementKind(me);
#MovementKind::#Complete;

#Receipt:#MovementKind;
#Reservation:#MovementKind;
#Release:#MovementKind;
#Shipment:#MovementKind;
```

Listing 30.1: Foundational stock-management vocabulary

The use of `#Complete` records that the four movement kinds are the complete vocabulary for this specification. If a project must allow extension by configuration, the vocabulary should not be declared complete.

30.4 Identified Temporal Business Entities

```
define .:#StockMovement(me)[
  product:#Product,
  warehouse:#Warehouse,
  kind:#MovementKind,
  quantity:#Quantity,
  occurredAt:#Time
];

define .:#StockPosition(me)[
  product:#Product,
  warehouse:#Warehouse,
  at:#Time,
  onHand:#Quantity,
  reserved:#Quantity,
  available:#Quantity
];

define .:#ReorderPolicy(me)[
  product:#Product,
  warehouse:#Warehouse,
  validAt:#Time,
  threshold:#Quantity,
```

```
  targetLevel:#Quantity
];
```

Listing 30.2: Movement, stock position and reorder policy entities

Time is present on movements, states and policies because their meaning changes over the lifecycle of the ERP system.

30.5 Derived Facts and Support Relations

```
define .:#ConsistentStockPosition(
  position:#StockPosition,
  product:#Product,
  warehouse:#Warehouse,
  at:#Time
);

define .:#BelowReorderThreshold(
  position:#StockPosition,
  product:#Product,
  warehouse:#Warehouse,
  at:#Time
);

define .:#ReorderRequired(
  product:#Product,
  warehouse:#Warehouse,
  at:#Time
);

define .:#InvalidStockPosition(
  position:#StockPosition,
  product:#Product,
  warehouse:#Warehouse,
  at:#Time
);

define .:#Subtract(left:#Quantity, right:#Quantity, result:#Quantity);
define .:#LessThan(left:#Quantity, right:#Quantity);
define .:#GreaterThan(left:#Quantity, right:#Quantity);
define .:#Negative(value:#Quantity);
```

Listing 30.3: Derived facts and arithmetic support relations

The arithmetic relations are support capabilities. They keep the rule vocabulary explicit and allow the implementation of numeric evaluation to remain a replaceable engine concern.

30.6 Movement Classification

The movement kind is stored as an attribute of an identified movement. Specialized conceptual classifications are derived from that value.

```
.:#StockMovement(movement)[kind=#Receipt]
-> .:#ReceiptMovement(movement);

.:#StockMovement(movement)[kind=#Reservation]
-> .:#ReservationMovement(movement);

.:#StockMovement(movement)[kind=#Release]
-> .:#ReleaseMovement(movement);

.:#StockMovement(movement)[kind=#Shipment]
-> .:#ShipmentMovement(movement);
```

Listing 30.4: Classification from the movement-kind attribute

This pattern lets a persistent business object retain one movement identity while acquiring conceptual classifications derived from its declared kind.

30.7 Stock Consistency and Reorder Propagation

```
.:#StockPosition(position)[
  product=product,
  warehouse=warehouse,
  at=t,
  onHand=onHand,
  reserved=reserved,
  available=available
]
and
.:#Subtract(onHand, reserved, available)
->
.:#ConsistentStockPosition(position, product, warehouse, t);
```

Listing 30.5: Consistent stock-position propagation

The reorder rule combines only facts that refer to the same product, warehouse and time.

```
.:#StockPosition(position)[
  product=product,
  warehouse=warehouse,
  at=t,
  available=available
]
and
.:#ReorderPolicy(policy)[
  product=product,
  warehouse=warehouse,
  validAt=t,
  threshold=threshold
]
and
.:#LessThan(available, threshold)
->
.:#BelowReorderThreshold(position, product, warehouse, t);

.:#BelowReorderThreshold(position, product, warehouse, t)
->
.:#ReorderRequired(product, warehouse, t);
```

Listing 30.6: Temporal reorder propagation

The intermediate fact is useful because it preserves the stock position that justified the recommendation. The final conclusion remains attached to the product, warehouse and time.

30.8 Invalid States as Explainable Facts

```
.:#StockPosition(position)[
  product=product,
  warehouse=warehouse,
  at=t,
  onHand=onHand
]
and .:#Negative(onHand)
-> .:#InvalidStockPosition(position, product, warehouse, t);

.:#StockPosition(position)[
  product=product,
  warehouse=warehouse,
  at=t,
  reserved=reserved
]
```

```
and .:#Negative(reserved)
-> .:#InvalidStockPosition(position, product, warehouse, t);

.:#StockPosition(position)[
  product=product,
  warehouse=warehouse,
  at=t,
  onHand=onHand,
  reserved=reserved
]
and .:#GreaterThan(reserved, onHand)
-> .:#InvalidStockPosition(position, product, warehouse, t);
```

Listing 30.7: Invalid stock-position deductions

A diagnostic fact can be displayed, queried, indexed, returned by a service and attached to its logical justifications. Whether it also blocks persistence is a separate application-policy decision made during model generation.

30.9 Derive Indexes from Query Patterns

The hypergraph already maintains indexes implied by relation slots. Additional `#Indexed` projections are justified when entity attributes must become relation slots or when a frequent access pattern needs a different leading order.

Representative stock queries include:

- movements by warehouse, time and kind;
- stock positions by product, warehouse and time;
- all stock positions in one warehouse at one time;
- reorder alerts by warehouse and time;
- invalid positions by warehouse and time.

```
define .:#StockPositionByWarehouseTimeProduct(
  warehouse:#Warehouse,
  at:#Time,
  product:#Product,
  position:#StockPosition
);

#StockPositionByWarehouseTimeProduct::#Indexed;

.:#StockPosition(position)[
  product=product,
  warehouse=warehouse,
  at=at
]
->
.:#StockPositionByWarehouseTimeProduct(
  warehouse,
  at,
  product,
  position
);
```

Listing 30.8: Attribute-to-tuple indexed projection

```
define .:#ReorderRequiredByWarehouseTimeProduct(
  warehouse:#Warehouse,
  at:#Time,
  product:#Product
```

```
);

#ReorderRequiredByWarehouseTimeProduct::#Indexed;

.:#ReorderRequired(product, warehouse, at)
->
.:#ReorderRequiredByWarehouseTimeProduct(
  warehouse,
  at,
  product
);
```

Listing 30.9: Indexed projection of a derived alert

The business fact and the index projection have different responsibilities. The first carries meaning; the second supplies an effective query path.

30.10 Manual Propagation Scenario

Assume the following values at time `T1`:

```
Product    = P1
Warehouse  = W1
onHand     = 10
reserved   = 4
available  = 6
threshold  = 7
```

Listing 30.10: Input fact values for the propagation scenario

The support capabilities establish:

```
.:#Subtract(10, 4, 6);
.:#LessThan(6, 7);
```

Listing 30.11: Arithmetic support facts

Expected logical results are:

```
.:#ConsistentStockPosition(Position1, P1, W1, T1);
.:#BelowReorderThreshold(Position1, P1, W1, T1);
.:#ReorderRequired(P1, W1, T1);
.:#ReorderRequiredByWarehouseTimeProduct(W1, T1, P1);
```

Listing 30.12: Expected inferred facts

For an invalid state with `onHand=5` and `reserved=7`, the expected result is:

```
.:#GreaterThan(7, 5);
.:#InvalidStockPosition(Position2, P1, W1, T1);
```

Listing 30.13: Expected invalid-state deduction

30.11 Human Review Checklist

Before accepting the ontology, a domain reviewer should confirm:

- whether stock positions and reorder policies require independent identity;
- whether the movement-kind vocabulary is truly closed;
- which time semantics apply to policies and movements;
- whether availability is stored, derived or both;

- whether invalid positions are accepted for diagnosis or rejected immediately;
- whether the proposed query orders correspond to real operational use;
- whether reorder recommendations require target quantity, lead time or supplier context.

The last item illustrates why the ontology remains an analysis artifact: new business questions can extend it before runtime files are generated.

30.12 LLM-Assisted Analysis Tasks

Given this ontology and its accepted rules, an LLM can safely assist with bounded tasks:

- identify terms in new requirements that are not defined in the ontology;
- propose an effective-time role when a statement is temporally ambiguous;
- suggest a diagnostic relation for an invariant;
- propose an index only after receiving a concrete query pattern;
- generate test facts and expected conclusions;
- prepare candidate logiCells model mappings.

It must label these outputs as proposals until a reviewer accepts them and the engine validates the resulting H-Logic.

30.13 Engine Test Plan

A complete validation campaign should test:

1. movement classification;
2. arithmetic relation evaluation;
3. temporal unification across position and policy;
4. reorder propagation;
5. invalid-state propagation;
6. indexed tuple creation;
7. repeated propagation and duplicate handling;
8. multiple justifications for one derived fact;
9. retraction after the last justification disappears;
10. absence of self-recreating projection cycles.

The ontology is ready for model generation only when the validation status is explicit: parser accepted, semantic resolution passed, propagation tests passed, or remaining capabilities documented as pending.

Chapter 31

From Validated H-Logic to Declarative Models

A validated H-Logic ontology describes executable meaning, but it is not yet a complete logiCells application. It belongs to the SEMANTIC side of the design. Generated YAML files are MODEL artifacts that still need their own declarative validation, while the assembled APPLICATION remains a separate step. This chapter explains how a validated semantic specification can guide source generation without collapsing these authorities into one.

31.1 Operationalization, Not Textual Translation

The transition can be summarized as:

`validated concepts and rules` → `generation decisions` → `candidate MODEL artifacts` → `declarative reparse / validation` → `APPLICATION compilation when required`

Several H-Logic constructs have natural model counterparts, but the mapping still requires decisions.

H-Logic meaning	**Candidate model artifact**	**Additional decision**
Identified type with `me`	Entity concept	ClassId, namespace and facets.
Pure polyadic relation	Relation concept	Reified persistence or graph-only fact.
Attribute	Model field	Type, size, nullability and rolePath.
Complete vocabulary	Enumeration-like concept set	Extension and migration policy.
Derived relation	H-Logic rule or generated relation	Materialization and lifecycle.
`#Indexed` projection	Queryable relation or runtime index	Storage, refresh and exposure strategy.
Diagnostic fact	Validation relation, event or view data	Whether it blocks persistence.

31.2 Generate Concepts from Identified Types

A foundational identified type:

```
define .:#Product(me);
```

Listing 31.1: Validated H-Logic type

can seed a logiCells model:

```
class:
  classId: lgc#product
  name: Product
  type: Entity

concepts:
  - Entity:
      facets:
        - main:
            type: hypergraph
            fields: []
```

Listing 31.2: Generated Product model skeleton

The namespace is a generation parameter. It must come from the target project conventions rather than being invented by the LLM.

31.3 Generate a Temporal Entity Model

The semantic source is:

```
define .:#StockPosition(me)[
  product:#Product,
  warehouse:#Warehouse,
  at:#Time,
  onHand:#Quantity,
  reserved:#Quantity,
  available:#Quantity
];
```

Listing 31.3: Validated StockPosition definition

A generated hypergraph facet can preserve the conceptual roles and attributes:

```
class:
  classId: lgc#stock_position
  name: StockPosition
  type: Entity

concepts:
  - Entity:
      facets:
        - main:
            type: hypergraph
            fields:
              - Product:
                  kind: slot
                  type: Object
                  target: lgc#product
                  rolePath: Product
              - Warehouse:
                  kind: slot
                  type: Object
                  target: lgc#warehouse
                  rolePath: Warehouse
              - At:
                  kind: attribute
                  type: DateTime
                  rolePath: At
              - OnHand:
                  kind: attribute
                  type: Decimal
                  rolePath: OnHand
```

```
            - Reserved:
                kind: attribute
                type: Decimal
                rolePath: Reserved
            - Available:
                kind: attribute
                type: Decimal
                rolePath: Available
```

Listing 31.4: StockPosition hypergraph facet generated from H-Logic

The H-Logic source does not decide whether `Available` should also be stored in a database facet. The generation policy must decide whether it is persisted, recomputed, cached or maintained by rules.

31.4 Generate Relations from Derived Facts

The validated relation:

```
define .:#ReorderRequired(
  product:#Product,
  warehouse:#Warehouse,
  at:#Time
);
```

Listing 31.5: Validated reorder conclusion

can become a graph relation model:

```
class:
  classId: lgc#reorder_required
  name: ReorderRequired
  type: Relation

concepts:
  - Relation:
      facets:
        - main:
            type: hypergraph
            fields:
              - Product:
                  kind: slot
                  type: Object
                  target: lgc#product
                  rolePath: Product
              - Warehouse:
                  kind: slot
                  type: Object
                  target: lgc#warehouse
                  rolePath: Warehouse
              - At:
                  kind: attribute
                  type: DateTime
                  rolePath: At
```

Listing 31.6: Generated ReorderRequired relation model

Whether instances are materialized permanently depends on the runtime strategy. A purely inferred relation can remain in conceptual memory, while a reporting or audit requirement may justify persistence.

31.5 Generation Decisions That H-Logic Does Not Make Alone

The generator must receive an explicit policy for concerns outside the semantic ontology.

```yaml
generationPolicy:
  namespace: lgc
  targetFormat: yaml

  concepts:
    StockMovement:
      persistence: database
      generateViews: true
    StockPosition:
      persistence: database
      generatedFields:
        Available:
          storage: derived-cache
    ReorderRequired:
      persistence: conceptual-memory
      publishAsService: true
    InvalidStockPosition:
      persistence: conceptual-memory
      blocksPost: false
      generateDiagnosticView: true

  indexes:
    generateFromIndexedTypes: true

  traceability:
    includeRequirementIds: true
    includeHLogicSourceLocations: true
```

Listing 31.7: Example generation policy

This policy keeps generator choices reviewable instead of embedding them invisibly in a skill prompt or procedural script.

31.6 Two Generation Modes

31.6.1 Generation driven by logiCells

A deterministic engine transformation uses validated H-Logic plus a generation policy:

$$\texttt{H-Logic} + \texttt{generation policy} \rightarrow \texttt{model generator} \rightarrow \texttt{YAML artifacts} \rightarrow \texttt{runtime validation}$$

This mode is appropriate for mappings that are stable, repeatable and fully specified. It should produce structured errors when a semantic construct has no supported model mapping.

31.6.2 Generation driven by LLM and skills

A specialized skill can generate richer initial artifacts:

- model and file names;
- manifests and package entries;
- initial views and services;
- comments and documentation;
- alternative runtime layouts.

The LLM remains bounded by validated inputs and target grammars. The generated artifacts must be compiled and checked by logiCells before acceptance.

$$\texttt{validated H-Logic} + \texttt{generation policy} + \texttt{project conventions} \rightarrow \texttt{LLM skill} \rightarrow \texttt{candidate YAML} \rightarrow \texttt{logiCells verification}$$

The skill is a generator, not the validator.

31.7 Verification of Generated Artifacts

Generated files pass through several checks:

1. YAML grammar validation;
2. manifest and package reference resolution;
3. classId and rolePath validation;
4. compilation of each model;
5. loading of the generated ClassItems;
6. execution of H-Logic propagation tests;
7. verification that generated persistence and views do not alter the accepted semantic behavior.

A syntactically valid model can still operationalize the wrong meaning. Acceptance tests must compare runtime behavior with the expected deductions established during H-Logic analysis.

31.8 End-to-End Traceability

Each generated artifact should retain a link to the requirement and H-Logic construct that justified it.

```yaml
traceability:
  - requirement: STOCK-RULE-003
    statement: Reserved stock cannot exceed physical stock.

    hlogic:
      source: stock-management.hlogic
      rule: reserved-greater-than-on-hand
      result: InvalidStockPosition

    generatedArtifacts:
      - models/StockPosition.model.yaml
      - models/InvalidStockPosition.model.yaml
      - views/InvalidStockPositions.view.xml

    validation:
      parser: passed
      semanticResolution: passed
      propagationTest: stock-invalid-003
      modelCompilation: passed
```

Listing 31.8: Requirement-to-runtime traceability record

This traceability supports review, regeneration, impact analysis and AI-assisted evolution.

31.9 A Four-Level Test Strategy

Parser tests Verify that the H-Logic source is accepted by the target grammar.

Semantic tests Verify type resolution, role compatibility, temporal unification and valid dependencies.

Propagation tests Verify expected facts, indexes, justifications and retractions.

Generated-model tests Verify compilation, loading and behavioral equivalence of the generated application artifacts.

For the stock example, a test with `onHand=10`, `reserved=4`, `available=6` and `threshold=7` must produce both `ConsistentStockPosition` and `ReorderRequired`. A test with `onHand=5` and `reserved=7` must produce `InvalidStockPosition`.

31.10 Recommended Project Artifacts

A project that uses this workflow can keep semantic analysis and generated application files separate:

```
specification/
  requirements/
    stock-management.md
  hlogic/
    stock-management.hlogic
  analysis/
    decisions.yaml
    questions.yaml
    traceability.yaml
  tests/
    parser/
    propagation/

generation/
  policy.yaml
  reports/

src/
  models/
  views/
  services/
  manifests/
  packages/
```

Listing 31.9: Suggested specification and generation layout

The specification directory is not an informal scratch area. It is the source of controlled meaning from which application artifacts can be regenerated.

31.11 Why This Workflow Matters

The approach changes the role of AI-assisted generation. Instead of asking an LLM to infer an application directly from prose, the project builds a verified semantic contract first.

$$\texttt{requirements} \rightarrow \texttt{controlled semantics} \rightarrow \texttt{verified inference} \rightarrow \texttt{generated declarative models} \rightarrow \texttt{executable application}$$

Humans validate business meaning, LLMs accelerate structured analysis and generation, and the LGC/logiCells engine remains the authority for syntax, semantic execution and model compilation.

Part VI

Building Applications

Chapter 32

Source Formats and Project Conventions

This edition uses one source convention for new development: **YAML for MODEL-level declarative metadata and XML only for visual `*.view.xml` sources**. The distinction is intentional. It makes the ownership of behavior and presentation visible in the file system and aligns new modules with the current logiCells compilation pipeline.

32.1 The source rule

For new modules, use:

Concern	Canonical source	Typical content
Business MODEL	`*.model.yaml`	entities, relations, facets, fields, actions, events, collections, states, constraints
Multi-class MODEL	`*.models.yaml`	several MODEL classes, especially VIEW controllers and object-behind classes
MODEL manifest	`_.manifest.yaml`	package-local class discovery and loading
Package	`*.package.yaml`	dependencies, schema directories, package items and services
VIEW package	`*.view.package.yaml`	ownership of a `meta/views` tree and its view items
Visual source	`*.view.xml`	frames, layouts, controls, datasets, data sources, cursors and explicit bindings to MODEL-behind classes
HTTP page resource	`*.lgp`	HTTP page-extension resource; route declared separately in `virtualPages`
Runtime launcher	YAML / `.lgc`	`kernel.config`, `kernel.abi`, Runtime paths, services and wallet settings
Application descriptor	`*.application.yaml`	package assembly, storage, models and services

MODEL XML can still exist in older projects and conversion tooling can read it, but it is

not taught as a source format for new MODEL development in this guide. When migrating an older application, preserve the legacy source long enough to establish behavioral equivalence, then publish the modern YAML representation.

32.2 YAML field convention

The field name is the YAML key. This guide uses the simplified field-key syntax consistently.

```yaml
fields:
  - Code: String
  - Customer:
      type: Object
      nullable: false
  - DisplayName:
      type: String
      rolePath: Customer.Name
      nullable: true
```

Listing 32.1: Preferred MODEL field syntax

32.3 VIEW is deliberately different

The VIEW source remains XML because the visual tree is hierarchical and the runtime VIEW grammar is XML-oriented. A modern view is not a monolithic form. It is an assembly of:

```
CustomerEdit.view.xml      # visual structure, datasets, view bindings
CustomerEdit.models.yaml   # Page / Master / component controllers
_.manifest.yaml   # discovery of the MODEL-behind classes
<module>.view.package.yaml # ownership of the view tree
```

Listing 32.2: Modern VIEW source split

The XML source references MODEL-behind classes explicitly through `source`, `class` and `classId`. The MODEL classes are compiled by the normal MODEL pipeline; VIEW compilation does not create a second MODEL compiler.

32.4 Migration rule for old forms and frames

Legacy `*.form` and `*.frame` files become `*.view.xml`. If an intermediate conversion has lost layout, bindings or controller structure, return to the original `.form`/`.frame` as the visual authority.

The migration is split into three independently verifiable results:

1. **visual parity**: significant layout, components, datasets and bindings survive in `*.view.xml`;
2. **MODEL-behind parity**: page/controller/model zones are externalized to `*.models.yaml` without losing behavior;
3. **binding parity**: every XML reference resolves to exactly one declared MODEL class and the intended `classId` is preserved.

32.5 HTTP Page-Extension Resources

The file extension for HTTP page resources is `.lgp`, for example `Users.lgp`. This is separate from the visual source `*.view.xml`, the Runtime launcher `.lgc`, and HTTP service descriptors. A virtual HTTP route is declared in a service definition under `virtualPages`; the existence of

a `.lgp` file does not activate an HTTP endpoint by itself. The service-level example appears in Chapter 48, “Declarative Runtime Services and Engine Patterns”.

32.6 Do not mix authorities

MODEL source is declarative source authority. APPLICATION is assembled metadata authority. SEMANTIC is the semantic graph/H-Logic authority. VIEW is visual-source authority plus explicit references to MODEL-behind classes. A tool or developer should always know which authority is being inspected or mutated.

Chapter 33

Class Manifests: Loading, Modes and Conventions

A class manifest is the runtime entry list for a model folder. It is not only a directory index. It tells the loader which conceptual classes are eligible to become runtime `ClassItem` objects, which identifiers they expose, and which local declarations belong to the package loading boundary.

33.1 What a Manifest Guarantees

A manifest gives the runtime an explicit loading contract:

```yaml
package:
  plugginCaption: TaskManagement
  uri: http://www.logicells.net/package/TaskManagement
  classes:
    - Task:
        classId: lgc#task
        type: Entity
```

Listing 33.1: Minimal class manifest in YAML

The presence of `Task.model` on disk is not sufficient. The class must also be exposed by the manifest and reached through an active package.

33.2 Manifest Attributes

Attribute	Role	Guidance
`classId`	Stable conceptual identifier	Treat it as a public identity. Do not change it casually after data, relations or integrations depend on it.
`name`	Runtime/public class name	Keep it aligned with the class declaration and complete class path used by collections and actions.
`type`	Broad class category	Use the category supported by the runtime declaration, such as entity or relation. Do not infer it only from the file name.

`fileName`	Physical source container	Optional for the ordinary one-class-per-file convention; required when a manifest entry is stored in a shared `*.models.yaml` container. It identifies the file, while the manifest item name selects the class inside that file.
`mode`	Optional loading or interpretation mode	Values are runtime-version dependent. Copy a value only from a validated model of the same generation and document why it is required. Absence of `mode` is preferable when the default behavior is sufficient.
`plugginCaption`	Descriptive package label	Use a stable human-readable package name. It helps inspection and traceability but does not replace package activation.
`uri`	Namespace/document identity	Use one stable URI convention per product or project. The URI organizes identity; the executable class identity still comes from `classId` and the loaded declaration.

33.3 Loading a Class from a Multi-Class .models Container

A `*.models.yaml` file is a physical container for several ordinary MODEL classes. The manifest keeps logical class identity separate from physical storage.

```yaml
package:
  uri: view#meta.views.Common.Folder
  classes:
    - FieldPanelSearch:
        classId: view#meta.views.Common.Folder.List.FieldPanelSearch
        fileName: List.models.yaml
        type: Entity
    - GridPanel:
        classId: view#meta.views.Common.Folder.List.GridPanel
        fileName: List.models.yaml
        type: Entity
```

Listing 33.2: Two manifest entries loaded from one models container

The loader resolves the manifest entry, opens `List.models.yaml`, selects the requested class from the `classes:` container by name, and sends that class through the standard MODEL loader. This is not a separate VIEW model format.

Each model folder that exposes loadable classes uses the manifest file `_.manifest.yaml`. The active package schema paths determine which manifests are loaded.

33.4 Choosing a Class Identifier Prefix

Platform concepts usually use `lgc#`. Application concepts should use a project-owned prefix rather than borrowing a platform prefix. The important properties are uniqueness, stability and ownership.

```yaml
platformConcept: lgc#entity
applicationConcept: tsk#task
```

```
3 applicationRelation: tsk#task_belongs_to_project
```

Listing 33.3: Platform and application identifier separation in YAML

A new prefix should be recorded in the project conventions and reused consistently across models, relations, tests and service contracts.

33.5 Duplicates and Cross-Folder References

A repeated class entry may be an accidental duplicate, a historical override, or a deliberate extension mechanism. Do not assume which one applies. Verify the effective runtime result:

1. compare the two declarations and their source paths;
2. identify which package and manifest are loaded first;
3. check whether the runtime reports replacement, extension or duplication;
4. resolve the class by `classId` and inspect its complete runtime name;
5. remove the duplicate unless the extension behavior is explicitly required and tested.

A manifest may reference declarations outside its immediate folder in older or specialized projects. For new code, prefer a local manifest that exposes local declarations and let package configuration compose folders. Cross-folder loading should be treated as an explicit architecture decision, not as an invisible shortcut.

33.6 Global Class Identifier Uniqueness

A `classId` is a conceptual identity in the assembled application, not a file-local name. Two independent non-atomic declarations must not silently reuse the same platform or application identifier.

For new modules, use this rule:

```
1 one meaning       -> one stable classId
2 another meaning   -> another classId
3 intentional replacement -> explicit redefinition mechanism
```

Listing 33.4: Class identity rule

For example, a business agent and an internal contextual participant should not share an identifier merely because both are called “agent” in natural language. Distinct concepts require distinct identities.

When a duplicate is detected, inspect the complete package and manifest loading chain before changing semantic roles. Historical projects may contain deliberate redefinitions, but those should be explicit and tested rather than inferred from load order.

33.7 Events in Manifests

Some historical manifests contain event declarations or loading hooks. These are not ordinary business events. They participate in package/model loading or specialization. For new modules, keep business events in the relevant model and use manifest-level behavior only when the runtime contract explicitly requires it.

33.8 Recommended Validation

```
1. The application loads the intended package.
2. The package item is active.
3. The manifest is found and compiled.
4. The model file is found and compiled.
5. The ClassItem is resolved by classId.
6. The resolved complete name matches the intended model.
```

Listing 33.5: Manifest validation sequence

When one of these steps fails, debug that layer before investigating views, services or business actions.

Chapter 34

Application Models

An application model is the boundary where domain concepts, runtime configuration, views, actions, services and packages are assembled into an executable application. It is not a thin project file. It describes how the conceptual runtime should load and interpret a coherent set of models.

34.1 The Role of an Application Model

A logiCells application model connects several concerns: the conceptual graph or hypergraph, the database aliases, the packages to load, the available services, the desktop or web publication profiles, the language and localization defaults, and the runtime containers.

The application model should not hide business semantics. Business concepts remain in `.model.yaml` or `.model.yaml` files. The application model assembles them.

34.2 Namespaces

Platform concepts usually use `lgc#`. Application examples may use `lbs#`. Both can coexist. The important rule is that each namespace has a clear responsibility: platform concepts provide reusable runtime semantics; application namespaces provide business-specific concepts.

34.3 Design Rules

Keep the application model focused on assembly. Put business objects in model files. Put reusable platform mechanisms in platform packages. Declare services and packages explicitly. Avoid mixing client-specific visual identity with reusable application architecture.

34.4 Summary

An application model is the executable assembly of a conceptual application. It binds packages, runtime resources, persistence and publication contexts without becoming the business model itself.

Chapter 35

Collections and Conceptual Views

Collections and views are two sides of the same navigation problem. A collection defines what can be traversed from a concept. A view defines how a slice of that traversal is presented to a user, service or agent.

35.1 Collections as Navigable Meaning

A collection should represent a meaningful access path: the phases of a work item, the messages of a conversation, the roles of a user, the items waiting in a queue. It should not be treated as an arbitrary SQL shortcut.

35.2 Views as Projections

A conceptual view projects a concept into an interaction. The view may use fields, lists, buttons and grids, but these are functional components. The concrete rendering is selected by registration of an implementation for the target platform.

35.3 Collections, Cursors and Context

When a view uses a `collectionPath`, it is asking the runtime to evaluate a collection in the current context. The context may be the current object, current cursor, current user or current process state.

35.4 Design Rules

Define collections using the vocabulary of the model. Use joins when the relationship itself must be navigable. Bind views to collections and cursors rather than duplicating data access code. Keep the visual appearance separate from the functional component declaration.

35.5 Summary

Collections make the model navigable. Views make navigation usable. Both should preserve conceptual meaning.

Chapter 36

Actions as Business Intentions

In logiCells, an action is not just a block of code. It is a declared capability of a concept. The action name, its owner, its parameters, its publication status and its routing rules express a business intention before they express an implementation technique.

This distinction is essential. A script may implement an action, but the script is not the action. A native component may implement an action, but the component is not the action. The action is the conceptual entry point that says what the concept can do.

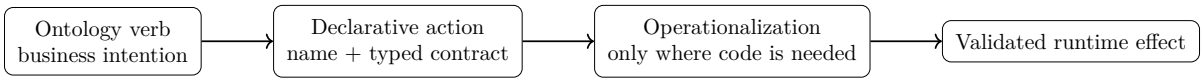


Figure 36.1: Code is pushed to the operationalization boundary; the business capability remains named and typed declaratively.

36.1 Why Actions Belong to Concepts

A business object should expose the operations that are meaningful for that object. A message may expose `NewMessage`. A conversation may expose `StartConversation`. A work item may expose `Validate`. A process may expose `Start`, `Finalize` or `Retry`.

The declaration keeps the intention visible:

```
actions:
- action:
    name: Start
    type: ObjectItem
    published: true
    params:
    - Result:
        dataType: Boolean
        paramType: ptReturn
```

Listing 36.1: Action declaration as business intention

The model says that the object can be started. The implementation decides how this start is performed.

36.2 Object-Level and Class-Level Actions

Object-level actions operate on a particular object item. They usually use `Self` as the current conceptual instance. They are appropriate when the behavior depends on the current object state.

Class-level actions operate on the concept as a whole. They are appropriate for search, creation, validation, role resolution or factory-like behavior. For example, a class-level action may find an existing object by name, create a new role if it does not exist or return the default network of an application namespace.

36.3 Scripted and Native Implementations

Many examples use scriptable actions. Scripted actions are useful because they keep orchestration near the model and make examples easy to read. However, logiCells does not require every action to be scripted. An action can also be implemented by a registered runtime class, service adapter, messaging component or visual component. When an action is implemented by a `functionalP` script, the script is only the implementation body. The practical beginner subset of that scripting language is introduced in Chapter 37, “functionalP for logiCells Beginners”.

The action declaration should therefore be read as a stable contract:

Here, `Verify` is the intention. The implementation class is the execution mechanism.

36.4 Parameters as Conceptual Boundaries

Parameters are part of the public meaning of an action. They define what the action expects and what it produces. A parameter may be a primitive type, such as `String` or `Boolean`, or a conceptual type such as `lgc#contact`, `lgc#conversation` or `lbs#work_item`.

Good action parameters make orchestration explicit. They also make the action easier to publish later as a service, call from a workflow or bind to a view component.

36.5 Published Actions

The `published` attribute is a visibility decision. A published action is part of the callable surface of the concept. An unpublished action is usually an internal helper, an event handler or a private implementation step.

Use published actions for stable business capabilities. Use unpublished actions for internal routing, view callbacks or intermediate process steps.

36.6 Actions in Processes

A process object coordinates actions on several participating objects. This gives a simple design rule:

```
If one object owns the behavior:
  declare the action on that object.

If several objects must collaborate:
  declare a process model and put the coordination action there.
```

Listing 36.2: When to place an action on a process

The process action may call local object actions synchronously or emit events asynchronously. In both cases, the coordination remains visible in the model.

36.7 Actions and Events

Actions and events form a pair. An event indicates that something meaningful happened or is requested. An action expresses what a concept does in response.

A process action may emit an event:

```
Event := Self.ObjectsContainer.BuildMessage('lbs.Process.Start');
Self.ObjectsContainer.SendEvent(Event, Self.EventProxy);
```

Listing 36.3: Action emitting an event

A controller may subscribe to that event and route it to an action:
This separation avoids hidden dependencies between the process and the controller.

Chapter 37

functionalP for logiCells Beginners

logiCells MODEL metadata is authored in YAML; visual VIEW sources use `*.view.xml`. Sometimes a model also needs a small executable fragment: initialize a page state, validate a dataset before it is posted, create an object, refresh a parent view or call a runtime action. In those places the metadata often uses `functionalP` scripts.

This chapter is not a complete language reference. It is the practical subset a beginner needs to read the scripts used in the first task-management tutorial and in dataset-driven views. The full language-oriented reference is in Chapter 91, "functionalP Language Essentials". A maintainer source map is retained in the repository but is intentionally excluded from the business-developer reading path.

37.1 Where functionalP Appears

A beginner usually meets `functionalP` in four places:

```
action script
  Implements a declared logiCells action.

pageModel event handler
  Reacts to page-level events such as OnShow.

dataset event handler
  Reacts to dataset lifecycle events such as BeforeDataSetOpened
  or BeforeDataSetPosted.

service or integration glue
  Calls a published capability or prepares a request/response boundary.
```

Listing 37.1: Common functionalP locations

The important rule is that `functionalP` is not the model syntax. The MODEL remains YAML. The script is an executable fragment owned by a declared action or event handler.

37.2 A Minimal Metadata Script

A script is normally stored inside an action. The surrounding YAML MODEL declares the action; the script provides the executable body.

In practice, many scripts do not explicitly assign `Result`. They perform an effect such as initializing a dataset or posting an object. For beginner code, use `Result` when the action has an expected return value. This makes intent easier to debug.

37.3 Variables and Assignment

The assignment operator is `:=`. Variables are often introduced by assignment:

```
lTitle := 'New task';
lPriority := 1;
Result := lTitle;
```

Listing 37.2: Local variables

Use local variable names that make runtime objects visible. A common convention is to prefix local variables with `l`: `lDs`, `lTask`, `lMem`, `lEvent`.

37.4 Self, Sender and Result

Most logiCells metadata scripts depend on three runtime names:

Name	Meaning in common scripts
`Self`	The current object behind the action or page. From `Self`, scripts usually reach `ObjectsContainer`, `PageControler` and the current conceptual object.
`Sender`	The object that triggered the event. In dataset events, `Sender` is often the dataset being opened, posted or refreshed.
`Result`	The explicit return value of a script when the action contract expects one.

A common beginner mistake is to use `Sender` as if it were always a visual component. In dataset events, it is normally a dataset. In visual events, it may be a visual component or its object-behind.

37.5 Accessing the PageControler

The `PageControler` is the local runtime object that connects the page model, the view data cache and the visual component tree.

```
lDs := Self.PageControler.DataSetsManager.DataSetByName('TaskEntityDs');
```

Listing 37.3: Accessing a page dataset through PageControler

Use `PageControler` when the script belongs to a page, a view, a component object-behind or a dataset-driven view event. Do not use it as a substitute for business modeling. It is a local coordination object for the current page.

37.6 Reading a Dataset

The `DataSetsManager` gives access to the datasets declared in the page view cache.

```
lDs := Self.PageControler.DataSetsManager.DataSetByName('TaskEntityDs');
lTask := lDs.ObjectItem;
```

Listing 37.4: Read the edited task dataset

This code does not query a database directly. It reads the current page data surface. The provider and collection behind the dataset remain outside the visual component.

37.7 Initializing Local Page State

The first task tutorial initializes a memory dataset when the page opens. This is a good beginner pattern because it shows the difference between business data and page state.

```
lMem := Self.PageControler.DataSetsManager.DataSetByName('TaskMemoryDs');
if lMem.IsEmpty then begin
  lMem.Append();
  lMem.OnlyOpen := True;
  lMem.PostData();
end;
```

Listing 37.5: Initialize a local filter dataset

`TaskMemoryDs` is not a persistent task table. It stores local filter state for the page. If the same value must survive outside the page, model it as a concept field or persistent facet instead.

37.8 Filtering Before a Dataset Opens

Dataset events allow a view to adjust its data surface before components read it.

```
if Sender.Name = 'TaskListDs' then begin
  Sender.SetOrdering('Priority desc, DueDate asc');
  lMem := Self.PageControler.DataSetsManager.DataSetByName('TaskMemoryDs');
  if lMem.OnlyOpen then
    Sender.AddStringFilterItem('StatusName','<>','Done');
  if lMem.SearchText <> '' then
    Sender.AddStringFilterItem('Title','contains', lMem.SearchText);
end;
```

Listing 37.6: Filter a dataset before opening

This script is a view-data rule. It should not contain core business policy. If the rule must be enforced everywhere, move it to a model constraint, action or service boundary.

37.9 Posting the Current Object

A save action usually reads the current dataset, retrieves its current object and posts it.

```
lDs := Self.PageControler.DataSetsManager.DataSetByName('TaskEntityDs');
lTask := lDs.ObjectItem;
lTask.PostData();
```

Listing 37.7: Post the current object

Posting data is an intentional runtime operation. A form field changing on screen does not necessarily mean the conceptual object has been persisted or propagated.

37.10 Creating an Object

Class-level or factory actions often create a new object through the object container.

```
lClassTask := Self.ObjectsContainer.ClassItemByName('TaskManagement.Task');
lTask := Self.ObjectsContainer.CreateObjectItem(lClassTask, 1, '');
lTask.SetValue('Title', 'New task');
lTask.PostData();
```

Listing 37.8: Create a new Task object

For production metadata, prefer a declared creation action when one exists. Direct object creation in a script is useful for tutorials and internal orchestration, but public workflows should expose stable actions.

37.11 Dispatching a Visual Event

After saving an object, a page may need to notify a parent view that a list should refresh.

```
lEvent := Self.PageControler.ParentPageControler.CreateVisualEvent(
  'lgc.Visual.View.Refresh', nil, nil,
  Self.ObjectsContainer.MetaDataContainer, false);
Self.PageControler.ParentPageControler.DispatchVisualEvent(lEvent, StopNotification);
```

Listing 37.9: Dispatch a parent visual refresh event

Use visual events for visual coordination. Do not use them as a hidden business event bus. If a business fact changed, represent it as a business action, event or process transition.

37.12 Calling a Visual Component Object-Behind

Some scripts need to call the local object-behind of a view component.

```
Self.PageControler.GetVisualComponentObjectBehindByName('Master').CloseForm();
```

Listing 37.10: Close the current master view

This is a local UI operation. It belongs in a page action, not in a domain model action intended for services or background execution.

37.13 When to Move Logic Elsewhere

A `functionalP` script should stay small. Move logic elsewhere when the script becomes a hidden application inside the metadata.

If the script...	**Move the logic to...**
Enforces a business rule everywhere	a model constraint, validation action or process step
Coordinates several objects over time	a workflow or process model
Publishes a stable external capability	a declared action and service contract
Only updates local view state	the page model or dataset event handler
Only refreshes or routes visual components	a visual event or reactive frame pattern

37.14 Beginner Mistakes

Mistake	**Correction**
Treating `functionalP` as the model language	Keep MODEL metadata in YAML. Use `functionalP` only inside declared executable fragments.
Putting business policy in a view event	Put durable policy in concepts, constraints, actions or workflows.
Assuming `Sender` always has the same type	Check the event family: dataset, visual, process, service or message.
Calling a component from a service action	UI object-behind calls belong to local page actions, not service boundaries.

Mistake	Correction
Forgetting to post data	Set values on the current object, then call `PostData()` when the change must be committed.
Leaving the return value implicit	Assign `Result` when the action contract expects a return value.

37.15 Practical Reading Path

For the first project, learn `functionalP` in this order:

1. read variables and assignment;
2. understand `Self`, `Sender` and `Result`;
3. access `PageControler.DataSetsManager`;
4. read the current dataset object;
5. post data;
6. dispatch a local visual event;
7. create an object only after the load and view paths are clear.

Chapter 38

Design Decision Guide: What to Create and When

This chapter consolidates the decision rules that were previously repeated across many chapters. Later chapters still explain the theory and the runtime details, but this guide gives the first practical answer to a beginner's question: *what should I create next?*

Use this chapter as a map. When a chapter says that concepts come before code, or that views are projections, or that events are observable changes, the operational question is answered here.

38.1 The Central Decision Chain

When starting a feature, do not begin with the screen, the table or the service route. Start with the smallest conceptual intention that must remain meaningful over time.

```
requirement
-> concept or relation
-> facets and fields
-> collections
-> view or API projection
-> actions
-> events
-> workflow, service or reactive frame if needed
-> tests and diagnostics
```

Listing 38.1: Practical design order for a new feature

The order is not a rigid waterfall. It is a sanity check. If a view, service or script becomes hard to explain, return to the concept, relation or action that should carry the meaning.

38.2 Choose the Construction Path Before the Artifacts

Before deciding what model artifact to create, decide how the project should move from business text to executable knowledge. logiCells supports two first-class starting paths, described in detail in Chapter 28, "Two Paths from Business Specifications to logiCells Applications".

- **Model-first**: choose this when the domain is sufficiently understood and the priority is to obtain a working declarative application quickly. Generate H-Logic later when a compact semantic synthesis or coherence check becomes useful.
- **H-Logic-first**: choose this when semantic ambiguity, interacting rules, heterogeneous vocabularies or formal consistency are major project risks. Validate the knowledge first, then generate model skeletons and operationalize the required business verbs as actions.

A project can move from one path to the other repeatedly. The decision is about the best *current feedback loop*, not about creating two incompatible application architectures. Model-first emphasizes feedback from the running application; H-Logic-first emphasizes feedback from semantic validation. Re-evaluate the choice whenever the dominant project risk changes.

38.3 Decision Table

Create this	**When**	**Avoid when**
Concept	The business vocabulary needs a stable thing with identity, fields, constraints, actions or views.	It is only a temporary UI value, a local filter, or a computed display value.
Relation	The meaning is between two or more participants and should be queryable, constrained or persisted as a fact.	A simple descriptive property is enough and the link has no independent meaning.
Facet	The same concept needs more than one projection: memory, persistence, search, import, external API, computed data.	The fields belong only to a view cache or a transient component state.
Field	The model needs a named position in a concept or facet. Use the project convention for slots and attributes.	The value is only a widget state such as “expanded”, “selected tab” or “typing text”.
Collection	The runtime must navigate, list, filter, join or publish a set of objects.	The data is a one-off object lookup that does not need a reusable navigation contract.
View	A concept or collection must be projected to a user, a component tree or a published UI.	The structure is only a back-end rule or service with no user-facing projection.
DataSet / DataSource / DataCursor	A view needs a local data cache, binding surface, navigation cursor, filter state or master/detail context.	The declaration belongs to the business model itself; model fields are not view-cache fields.
Action	The system must express an intention: create, validate, post, refresh, publish, compute, notify or execute.	The behavior is only automatic propagation that can be handled by a dataset event or view binding.
Event	Something observable should trigger a reaction: UI click, dataset lifecycle, business change, process transition, message arrival.	You only need a direct action call and no observer, routing or propagation boundary exists.
Workflow / Process	Several actions must be coordinated with state, ordering, human steps, retries, asynchronous behavior or traceability.	A single atomic action is enough.
Service	A model capability must be exposed outside the local runtime or UI through a stable publication boundary.	You are exposing an internal helper or a script that should remain private.

Create this	When	Avoid when
Reactive desktop frame	A page needs a dynamic host region, child routing, parent/child event bubbling, preview, workflow shell or dataset-aware visual coordination.	A simple form, grid, split view or tabbed view is sufficient.
AI assistant	The model should be read, explained or acted on through controlled conceptual actions and traceable context.	The assistant would bypass model semantics or call private implementation details directly.

38.4 Quick Questions Before Creating Anything

Ask these questions before adding a new artifact:

- Is this a domain idea that should survive UI or database changes? If yes, create or extend a concept or relation.
- Is this a representation of an existing concept for storage, memory, search or API access? If yes, create or extend a facet.
- Is this a reusable list or traversal contract? If yes, create a collection.
- Is this a user or system intention? If yes, create an action.
- Is this a reaction to something observable? If yes, create or subscribe to an event.
- Is this a multi-step coordination problem? If yes, create a workflow or process model.
- Is this a UI composition problem? If yes, choose a view component, DataSet cache or reactive frame.
- Is this an external boundary? If yes, publish a service or API adapter intentionally.

38.5 Common Beginner Decisions

38.5.1 Concept or Field?

Create a field when the value describes an existing concept. Create a concept when the value has its own identity, behavior, lifecycle, constraints or views. For example, `Priority` can be a field on `Task`. A `Project` should usually be a concept because it groups tasks, has its own lifecycle and may have its own views.

38.5.2 Relation or Foreign Key?

A database foreign key is a storage projection. A logiCells relation is conceptual meaning. Use a relation when the link itself matters: membership, assignment, ownership, dependency, delegation, approval or temporal participation.

38.5.3 Action or Event?

An action says what the system intends to do. An event says what happened or what should be observed. A button usually triggers an action. A dataset post, a selection change or a business state transition may emit or dispatch an event.

38.5.4 Action or Workflow?

Use an action for a single intention. Use a workflow when the intention has steps, state, routing, human interaction, asynchronous behavior, retries or audit requirements.

38.5.5 View or Reactive Frame?

Use a normal view for a stable projection such as a form, list or dashboard block. Use a reactive desktop frame when the area hosts dynamic children, routes local events, reacts to dataset changes or acts as a workflow shell.

38.6 Bad Pattern, Correction Pattern

Bad pattern	Why it hurts	Correction
Start from a screen and invent fields as the UI grows.	The concept becomes accidental and cannot be reused consistently.	Name the concept first, then declare facets, collections and views.
Put business meaning only in a script.	The runtime cannot inspect, publish or govern the intention.	Declare the action, relation or constraint; let scripts implement the behavior.
Expose every action as a service.	Internal helpers become public contracts.	Publish only stable capabilities with explicit inputs, outputs and security expectations.
Use a dataset field as if it were a model field.	View cache and domain model become confused.	Keep model fields in facets; keep DataSet fields in the view cache.
Create a workflow for a single atomic operation.	The model becomes heavier than the behavior.	Use an action, then promote it to a process only when coordination state is needed.
Route all UI events through the server.	The interface becomes slow and over-coupled.	Use local PageControler dispatch for local visual coordination; cross the boundary only for model or service changes.
Copy legacy package names into public examples.	Documentation leaks project history and confuses beginners.	Use neutral namespaces and describe the reusable pattern.

38.7 Where to Continue

For the core beginner path, continue through the application-building chapters in this guide. For View cache, DataSet binding, local visual-event routing and advanced reactive interfaces, continue in the companion *logiCells Rich Client View Guide*.

Chapter 39

Events as Observable Conceptual Changes

Events are the observable side of conceptual change. They describe that something has happened in the conceptual runtime: an object was selected, a frame was shown, a process started, a process completed, a message was received, or a business state changed.

In logiCells, events are not merely callbacks. They are declared in metadata. They can carry parameters, bind those parameters to actions, restrict their source, and route the reaction to a specific object, class or controller.

39.1 Events and Actions

The basic pattern is a declaration that maps an event class to an action path. Parameters are copied from the event into the action call.

```
events:
  - lbs.ServiceVisit.Completed:
      sources: lbs#service_visit_process/*
      actions:
        - action:
            path: ./Action[@name='OnServiceVisitCompleted']
            params:
              - Source:
                  eventParam: Source
              - Sender:
                  eventParam: Sender
```

Listing 39.1: Event-to-action binding

This declaration keeps the reactive behavior visible. The application does not hide the rule in a programming framework. The model states that a particular event triggers a particular action.

39.2 Local and Remote Events

Some events are local to a view or runtime object. Others are intended to cross process, component or service boundaries. A model can distinguish events that are local, remote, or both.

```
events:
  - App.Conversation.Open:
      isLocal: true
      isRemote: true
      params:
        - Sender:
```

```
            dataType: Object
            paramType: ptIn
```

Listing 39.2: Local and remote event declaration

This pattern is central for asynchronous applications. The same conceptual event can drive a local UI reaction and also participate in distributed messaging.

39.3 Events as Process Boundaries

A process should emit events at its business boundaries: start, waiting, validation, completion, failure and cancellation. These events are the contract observed by controllers, views, services and other processes.

```
actions:
- action:
    name: Start
    class: ObjectItemScriptableObjectElementAction
    type: ObjectItem
    published: true
    scripts:
    - clientScript: true
      serverScript: false
      language: functionalP
      classes: Desktop
      cdata: |
        Event := Self.ObjectsContainer.BuildMessage('lbs.Process.Started');
        Self.ObjectsContainer.SendEvent(Event, Self.EventProxy);

- action:
    name: Finalize
    class: ObjectItemScriptableObjectElementAction
    type: ObjectItem
    published: true
    scripts:
    - clientScript: true
      serverScript: false
      language: functionalP
      classes: Desktop
      cdata: |
        Event := Self.ObjectsContainer.BuildMessage('lbs.Process.Completed');
        Self.ObjectsContainer.SendEvent(Event, Self.EventProxy);
```

Listing 39.3: Process event boundaries

The important point is not the specific event name. The important point is that the process exposes meaningful lifecycle events.

39.4 Controllers as Event Observers

A controller should observe process events and update the application experience. It should not own the business process itself.

```
class:
  classId: lbs#application_controller
  name: ApplicationController
  type: Entity

  events:
    - lbs.Process.Started:
        sources: lbs#service_visit_process/*
        actions:
          - action:
              path: ./Action[@name='OnProcessStarted']

    - lbs.Process.Completed:
```

```
14          sources: lbs#service_visit_process/*
15          actions:
16            - action:
17                path: ./Action[@name='OnProcessCompleted']
```

Listing 39.4: Controller observing process events

This separation makes the architecture easier to reason about. The process owns the business progression. The controller owns the application reaction.

39.5 Visual Events

Views and frames can also declare events. A button may emit a visual action event, a frame may emit an on-show event, and a parent frame may receive navigation events.

Visual events should remain visual. They may ask a process to start or ask a controller to navigate, but they should not contain the domain workflow itself.

Chapter 40

Workflows and Processes

A logiCells workflow is not only a sequence of screens or a piece of procedural code. It is an executable conceptual object. It may hold state, reference business objects, expose actions, react to events, and coordinate several participants over time.

This distinction is important because applications usually become complex at the boundaries between objects. A business object can own its local data and local behavior, but a cross-object operation needs a place where the coordination is explicit. In logiCells, that place is a process model.

40.1 Process as a Conceptual Object

A process model should be introduced when an operation is no longer local to one object. Typical signs are:

- several business objects participate in the same operation;
- the operation has a visible lifecycle, such as started, waiting, validated, completed or failed;
- the operation must be resumed after an external event;
- the user interface must react to business progress;
- the coordination must be logged, audited or tested independently.

A process model does not replace the participating objects. It references them and coordinates them. Business objects remain responsible for their own consistency. The process model is responsible for the order, the state and the interaction pattern.

40.2 Synchronous and Asynchronous Coordination

A process can coordinate objects in two complementary ways.

In a synchronous process, one action invokes a sequence of other actions directly. The script acts as an explicit orchestration. This is appropriate when the whole operation must run immediately and deterministically.

In an asynchronous process, actions emit events or messages. Other objects listen to those events and react when they are ready. This is appropriate when the process crosses UI boundaries, remote services, message queues, devices, users or long-running operations.

Both patterns need a process model. Without a process model, synchronous orchestration becomes hidden in arbitrary scripts, and asynchronous orchestration becomes scattered across listeners. With a process model, the coordination remains visible in the metadata.

40.3 The Role of `lgc#process`

The platform defines the notion of process as a reusable conceptual root. A process can expose an `Execute` action for direct execution and an `Infer` action for state progression or rule-based evolution. Application-specific processes may imply or specialize this root when they need a reusable coordination contract.

The following pattern is recommended:

```
class:
  classId: lbs#construction_site_process
  name: ConstructionSiteProcess
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: lgc#process
        facets:
          - memory:
              type: hypergraph
              fields:
                - ConstructionSite: lbs#construction_site
                - CurrentState: String
```

Listing 40.1: Process specialization pattern

The prefix `lgc#` identifies reusable platform concepts. The prefix `lbs#` may be used for example or application-domain concepts. An application prefix should remain clearly separated from platform prefixes.

40.4 Process Models as Anti-Complexity Boundaries

The architectural rule is simple:

> As soon as a process goes beyond a local action, it should become an explicit process model.

This rule avoids three common forms of accidental complexity.

First, it avoids putting cross-object logic inside one business object. A construction site, a user or a message should not secretly orchestrate the whole application.

Second, it avoids putting domain workflow inside views. A view may send an event or call a process action, but the process should own the business progression.

Third, it avoids distributing state transitions across unrelated event handlers. Event handlers should react to events, while the process model should define what those reactions mean in the overall flow.

40.5 Synchronous Process Example

The following simplified process coordinates object creation synchronously. The process creates a business object, creates a related voucher, persists both, then records the final state.

```
class:
  classId: lbs#service_visit_process
  name: ServiceVisitProcess
  className: CustomClassItem
  currentGeneratorName: Demo.Process.ServiceVisitProcess
  type: Entity

  object:
    className: CustomObjectItem
    objectName: ServiceVisitProcess
```

```
11
12    concepts:
13    - Entity:
14        constraints:
15        - implies: lgc#process
16        facets:
17        - memory:
18            type: hypergraph
19            fields:
20            - WorkSite: lbs#work_site
21            - Customer: lbs#customer
22            - Voucher: lbs#service_voucher
23            - CurrentState: String
24
25    actions:
26    - action:
27        name: Execute
28        class: ObjectItemScriptableObjectElementAction
29        type: ObjectItem
30        published: true
31        params:
32        - Result:
33            dataType: Boolean
34            paramType: ptReturn
35        scripts:
36        - clientScript: true
37          serverScript: false
38          language: functionalP
39          classes: Desktop
40          cdata: |
41            Result := false;
42            Voucher := Self.ObjectsContainer.CreateObjectItem('lbs#service_voucher', 1, '');
43            Voucher.Customer := Self.Customer;
44            Voucher.WorkSite := Self.WorkSite;
45            Voucher.PostData();
46
47            Self.Voucher := Voucher;
48            Self.CurrentState := 'Completed';
49            Self.PostData();
50            Result := true;
```

Listing 40.2: Synchronous process model in YAML

Here, the process action is intentionally synchronous. It is easy to test and easy to trace, because the process object owns the orchestration.

40.6 Asynchronous Process Example

The same kind of process can be modeled asynchronously. The process starts by emitting an event. Other objects or controllers may react. When the expected response arrives, the process finalizes itself.

```
1  class:
2    classId: lbs#service_visit_process
3    name: ServiceVisitProcess
4    type: Entity
5
6    concepts:
7    - Entity:
8        constraints:
9        - implies: lgc#process
10       facets:
11       - memory:
12           type: hypergraph
13           fields:
14           - WorkSite: lbs#work_site
15           - Voucher: lbs#service_voucher
16           - CurrentState: String
17
```

```yaml
events:
- lbs.ServiceVisit.Validated:
    sources: lbs#service_visit_process/*
    actions:
    - action:
        path: ./Action[@name='Finalize']

actions:
- action:
    name: Start
    class: ObjectItemScriptableObjectElementAction
    type: ObjectItem
    published: true
    scripts:
    - clientScript: true
      serverScript: false
      language: functionalP
      classes: Desktop
      cdata: |
        Event := Self.ObjectsContainer.BuildMessage('lbs.ServiceVisit.Started');
        Self.ObjectsContainer.SendEvent(Event, Self.EventProxy);
        Self.CurrentState := 'Started';

- action:
    name: Finalize
    class: ObjectItemScriptableObjectElementAction
    type: ObjectItem
    published: true
    scripts:
    - clientScript: true
      serverScript: false
      language: functionalP
      classes: Desktop
      cdata: |
        Self.CurrentState := 'Completed';
        Self.PostData();
        Event := Self.ObjectsContainer.BuildMessage('lbs.ServiceVisit.Completed');
        Self.ObjectsContainer.SendEvent(Event, Self.EventProxy);
```

Listing 40.3: Asynchronous process model in YAML

The process no longer assumes that every step occurs in the same call stack. It coordinates the flow through events.

40.7 Design Rules

Business objects own local behavior. Process models coordinate cross-object behavior. Events make coordination observable. States make the progression understandable. Rules make the progression valid. Views should display and trigger the process, but not own the process.

When the coordination has business meaning, give it a model. When the coordination is only a visual reaction, keep it in the view or controller. When the coordination crosses local and remote boundaries, use events or messaging.

Chapter 41

Asynchronous Messaging as a Platform Pattern

Asynchronous coordination is a platform capability in logiCells. The messaging layer provides reusable concepts for messages, conversations, participants, message states and event dispatch. Application models can use those concepts to coordinate distributed or delayed interactions without embedding transport logic in business objects or views.

41.1 Platform Messaging Concepts

The reusable platform pattern includes:

- message concepts such as `lgc#custom_message` and `lgc#text_message`;
- workflow message concepts such as `lgc#adhoc_workflow_message`;
- conversation concepts such as `lgc#conversation`;
- participant concepts used to connect actors to conversations;
- message creation, message states and timestamps;
- event construction and event dispatch;
- local and remote event delivery;
- package-level integration with the messaging platform.

The purpose of this chapter is to describe those reusable mechanisms as part of the logiCells execution model.

41.2 Message as a Conceptual Object

A message is a model object. It may have identity, class identity, creation date, reception date, sender name, a conversation reference and a state.

```
class:
  classId: lgc#custom_message
  name: CustomMessage
  className: CustomClassItem
  namespace: Messaging
  currentGeneratorName: Messaging.CustomMessage
  type: Entity

```

```
  object:
    className: CustomObjectItem
    objectName: CustomMessage

  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
        facets:
          - CustomMessage:
              type: database
              map: lgcCustomMessage
              fields:
                - IdCustomMessage:
                    type: String
                    rolePath: Id
                    size: 32
                - ClassIdCustomMessage:
                    type: String
                    rolePath: ClassId
                    defaultValue: lgc#custom_message
                - IdConversation:
                    type: String
                    rolePath: Conversation.Id
                    nullable: true
                - CreationDate:
                    type: DateTime
                - ReceptionDate:
                    type: DateTime
                - SenderName:
                    type: String
```

Listing 41.1: Reusable custom message pattern

This is not a transport packet. It is a conceptual object that can be persisted, queried, routed and attached to conversations.

41.3 Message States

A message can define states. The exact state vocabulary may evolve, but the pattern is stable: a message is not only a payload; it is also a stateful conceptual object.

```
states:
  entityStateIndexFieldName: StateIndex
  items:
    - Created:
        stateIndex: 0
        description: Message has been created
    - Empty:
        stateIndex: 1
        description: Message has no useful content yet
    - Sent:
        stateIndex: 2
        description: Message has been sent
    - Received:
        stateIndex: 3
        description: Message has been received
```

Listing 41.2: Message state pattern

States make asynchronous behavior observable. They also make retry, acknowledgement, audit and traceability easier to model.

41.4 Event Dispatch from a Message

The asynchronous pattern appears when a message action builds an event and dispatches it.

```
actions:
- action:
    name: NewMessage
    class: ObjectItemScriptableObjectElementAction
    type: ObjectItem
    published: false
    params:
    - Sender:
        dataType: Object
        paramType: ptIn
    scripts:
    - clientScript: true
      serverScript: false
      language: functionalP
      classes: Desktop
      cdata: |
        Event := Self.ObjectsContainer.BuildMessage('lbs.Message.Receive');
        Event.Sender := Sender;
        Self.ObjectsContainer.SendEvent(Event, Self.EventProxy);
```

Listing 41.3: Message-triggered event dispatch

The message object remains part of the conceptual model. The event expresses the asynchronous change that other concepts may observe.

41.5 Conversation as Coordination Context

A conversation gives messages a coordination context. It lets the platform group exchanges, associate participants and support asynchronous interaction over time.

```
actions:
- action:
    name: StartConversation
    class: ObjectItemScriptableObjectElementAction
    type: ObjectItem
    published: true
    params:
    - Sender:
        dataType: Object
        paramType: ptIn
    scripts:
    - clientScript: true
      serverScript: false
      language: functionalP
      classes: Desktop
      cdata: |
        ConversationCI := Self.ObjectsContainer.ClassItemByClassId('lgc#conversation');
        Conversation := Self.ObjectsContainer.CreateObjectItem(ConversationCI, 1, '');
        Conversation.Name := Self.Name;
        Conversation.PostData();

        ParticipantsCI := Self.ObjectsContainer.ClassItemByClassId('lgc#participants');
        Participant := Self.ObjectsContainer.CreateObjectItem(ParticipantsCI, 1, '');
        Participant.Conversation := Conversation;
        Participant.Participant := Sender;
        Participant.PostData();
```

Listing 41.4: Starting a conversation from a contact-like object

The domain object initiates a conversation, but the conversation and participant concepts remain reusable platform concepts.

41.6 Local and Remote Messaging

A model may declare events as local, remote or both. This allows the same conceptual event to be handled by the local runtime and by a remote messaging layer.

```
events:
  - lbs.Conversation.Open:
      isLocal: true
      isRemote: true
      params:
        - Sender:
            dataType: Object
            paramType: ptIn
```

Listing 41.5: Local and remote conversation event

The capability to route events locally and remotely belongs to the logiCells platform. Application namespaces such as `lbs#` can use that capability without redefining the messaging infrastructure.

41.7 Packaging Pattern

A complete application may load several packages. Platform packages provide kernel, services, controllers, visuals and messaging. Application packages provide domain models, views, resources and optional product-specific extensions.

```
packages:
  - package:
      path: '[BasePath]/packages/Common.package'
      caption: Kernel
      enabled: true
      groupId: lgc#application_group
  - package:
      path: '[BasePath]/packages/CommonServices.package'
      caption: Kernel Services
      enabled: true
      groupId: lgc#application_group
  - package:
      path: '[BasePath]/packages/messagingPlatform.package'
      caption: Messaging Platform
      enabled: true
      groupId: lgc#application_group
  - package:
      path: '[BasePath]/packages/BusinessApplication.package'
      caption: Business Application
      enabled: true
      groupId: lbs#application_group
```

Listing 41.6: Package separation pattern

The important point is that messaging remains reusable infrastructure. Application packages consume it through declared message, conversation and event models.

Chapter 42

Roles, Permissions and Contextual Visibility

Security in a conceptual application is not limited to hiding buttons. It governs which concepts, views, actions and services are visible or executable in a given context.

42.1 Roles as Conceptual Positions

A role describes what an actor may do in relation to a model. Some roles are platform roles. Others belong to an application namespace such as `lbs#`. A role may control view visibility, action publication, process transitions or service access.

42.2 Permission Boundaries

Published actions require special attention. If an action can be reached through a webservice, agent or UI, its authorization must be explicit. The action signature is public; the execution must still be controlled by context.

42.3 Contextual Visibility

Visibility depends on the current user, object, process state, publication channel and sometimes the device or runtime mode. A validation action may be visible only while a process is waiting for approval. A service action may be callable only by an integration role.

42.4 Design Rules

Do not rely on UI visibility as the only security rule. Protect published actions. Treat webservice access as a first-class publication channel. Keep role checks close to the action or process they protect.

42.5 Summary

Roles and permissions make conceptual applications governable. They should apply consistently across views, services, processes and agents.

Chapter 43

Application Modeling Design Rules

Good logiCells applications are built from stable modeling rules. The following rules summarize the patterns used throughout this guide.

43.1 Separate Responsibilities

Use business objects for local data and behavior. Use relation objects when a relation needs identity, state or attributes. Use process objects when several business objects must be coordinated. Use controllers for application orchestration. Use views as projections, not as workflow owners.

43.2 Model Before Code

Prefer declaring concepts, roles, facets, actions, events and views before writing implementation code. Scripts and registered classes should implement a declared intention, not invent hidden model semantics.

43.3 Publish Intentionally

A published action is a capability contract. It may be consumed by UI, webservice, agent or process. Do not publish helper actions that should remain internal.

43.4 Pair Examples

Use YAML for MODEL examples. Use XML examples only when explaining a canonical `*.view.xml` visual source.

43.5 Summary

Modeling discipline keeps the system understandable: concepts own meaning, processes own coordination, events own observability, and publication exposes stable capabilities.

Chapter 44

Runtime Objects, Containers, and Execution Contexts

A logiCells model becomes useful when it enters an execution context. A `.model` file declares classes, concepts, facets, actions, collections, events, and views. The runtime turns these declarations into objects that can be created, queried, linked, executed, persisted, and observed.

The central runtime idea is the *conceptual container*. A container is not merely a technical object registry. It is the execution boundary inside which conceptual identifiers, class definitions, object instances, collections, event routing, scripts, and persistence projections become coherent.

44.1 From Declaration to Runtime Object

A class declaration introduces a conceptual type, usually identified by a stable `classId`. At runtime, this declaration is represented by a class object. Instances created from that class become object items.

```
class:
  classId: lgc#work_item
  name: WorkItem
  type: Entity
  object:
    className: CustomObjectItem
    objectName: WorkItem
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
        facets:
          - main:
              type: hypergraph
              fields:
                - Title: String
                - Status: String
```

Listing 44.1: Runtime reading of a declarative model

This declaration can be read in three layers. At the modeling layer, it defines an entity named `WorkItem`. At the memory layer, it introduces a conceptual structure with roles such as `Title` and `Status`. At the runtime layer, it makes it possible to create, retrieve, modify, persist, and display work item objects.

44.2 Class Items and Object Items

logiCells separates the model of an object from the object itself.

Class item The runtime representation of a conceptual class. It owns the class identifier, generators, actions, collections, views, and metadata.

Object item A runtime instance of a conceptual class. It carries values, relations, state, and executable object-level behavior.

In script examples, this separation is commonly visible through operations such as looking up a class by its identifier and then creating or retrieving an instance.

```
WorkItemClass := Self.ObjectsContainer.ClassItemByClassId('lgc#work_item');
WorkItem := Self.ObjectsContainer.CreateObjectItem(WorkItemClass, 1, '');
WorkItem.Title := 'Prepare field intervention';
WorkItem.Status := 'Created';
WorkItem.PostData();
```

Listing 44.2: Typical runtime lookup and object creation

The important point is not the syntax of the script. The important point is the architectural pattern: declarative metadata gives the runtime enough information to construct and execute conceptual objects dynamically.

44.3 The Objects Container

The objects container is the runtime environment in which object operations are resolved. It can find a class, create an object, find an existing object, route an event, build a message, open a collection provider, or access the metadata container.

Typical responsibilities include:

- resolving conceptual class identifiers such as `lgc#entity`, `lgc#process`, or `lbs#work_item`;
- creating instances through metadata rather than hard-coded constructors;
- providing access to object collections and joins;
- supporting script execution inside an object context;
- dispatching local or asynchronous events;
- synchronizing object values with persistence projections.

The container is also the place where platform and application namespaces meet. A platform concept may use a `lgc#` identifier, while an application-specific concept may use a namespace such as `lbs#`. The runtime does not require everything to belong to the same namespace. It requires the identifiers to be stable and the model graph to remain coherent.

44.4 Self, Context, and Runtime Binding

When an action is executed, `Self` denotes the current object or class receiving the action. This makes actions reusable: the same declarative action pattern can operate on different instances because it is bound at execution time.

```
action:
  name: MarkAsCompleted
  type: ObjectItem
  scripts:
    - language: functionalP
      cdata: |
        Self.Status := 'Completed';
        Self.PostData();
```

Listing 44.3: Object-level action pattern

The same object can also use the container to coordinate other objects.

```
Voucher := Self.ObjectsContainer.CreateObjectItem('lbs#service_voucher', 1, '');
Voucher.WorkItem := Self;
Voucher.PostData();
```

Listing 44.4: Creating a related object from an action

This is the first step toward process modeling: once an action starts coordinating several objects, the coordination usually deserves its own process model.

44.5 Persisting Conceptual Changes

A logiCells object can exist in conceptual memory, in a persistent projection, or in both. A database facet describes how conceptual roles are projected into records and fields. Runtime calls such as `PostData` make the change durable according to the projection rules declared in the model.

```
facets:
  - main:
      type: hypergraph
      fields:
        - Title: String
        - Status: String
  - WorkItem:
      type: database
      map: lbsWorkItem
      fields:
        - IdWorkItem:
            rolePath: Id
        - ClassIdWorkItem:
            rolePath: ClassId
            defaultValue: lbs#work_item
        - Title:
            size: 255
        - Status:
            size: 32
```

Listing 44.5: Hypergraph and database facets for the same concept

The hypergraph facet expresses conceptual structure. The database facet expresses persistence. Keeping them together in the same model makes the runtime object both meaningful and operational.

44.6 Design Rule

The practical rule is simple:

> A model describes what an object means; the container gives that model an execution context.

When this rule is respected, application code does not need to hard-code every class, relation, workflow, or view. The runtime can discover them from the conceptual model.

Chapter 45

Collections, Joins, and Queryable Models

A logiCells application is not only a set of isolated objects. It is a navigable conceptual space. Collections and joins define how objects can be retrieved, projected, filtered, and combined.

Collections are therefore part of the model, not merely part of the user interface. A collection can expose all instances of a class, a filtered subset, the content of a relation, or a join between several conceptual sources.

45.1 Collections as Runtime Navigation Contracts

A collection declaration gives a name to a way of navigating objects.

```
collections:
  - collection:
      name: Entity
      target: ObjectItem
      providerFactory: ObjectItemInstancesCollectionProviderFactory
```

Listing 45.1: Simple collection provider pattern

The consumer of this collection does not need to know how the objects are stored. It only needs to know that the conceptual class exposes a collection named `Entity`. The runtime resolves the provider and returns a navigable set of object items.

45.2 Relations as Queryable Structures

A relation model often appears as a normal entity with fields pointing to the related concepts. The database facet stores the relation keys, while the hypergraph facet keeps the conceptual roles visible.

```
class:
  classId: lbs#work_item_phase
  name: WorkItemPhase
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
        facets:
          - main:
              type: hypergraph
              fields:
                - WorkItem: lbs#work_item
                - Phase: lbs#phase
          - WorkItemPhase:
```

```
            type: database
            map: lbsWorkItemPhase
            fields:
              - IdWorkItemPhase:
                  rolePath: Id
              - IdWorkItem:
                  rolePath: WorkItem.Id
              - ClassIdWorkItem:
                  rolePath: WorkItem.ClassId
                  defaultValue: lbs#work_item
              - IdPhase:
                  rolePath: Phase.Id
              - ClassIdPhase:
                  rolePath: Phase.ClassId
                  defaultValue: lbs#phase
```

Listing 45.2: Relation object linking a work item and a phase

This style makes relations first-class. A relation can have its own identity, fields, constraints, actions, history, and views.

45.3 Join Collections

A join collection projects information from several sources into a single navigable result. It is useful when a view must show a business object together with information coming from related objects.

```
collections:
  - join:
      name: WorkItemPhases
      target: ObjectItem
      fields:
        - Id:
            fieldName: WorkItemPhase.IdWorkItem
        - ClassId:
            fieldName: WorkItemPhase.ClassIdWorkItem
        - IdPhase:
            fieldName: Phase.Id
            rolePath: Phase.Id
        - PhaseName:
            fieldName: Phase.Name
      collections:
        - collection:
            name: WorkItemPhase
            target: ClassItem
            sourcePath: /ClassItem[@classId='lbs#work_item_phase']/Collection[@name='Entity']
        - collection:
            name: Phase
            target: ClassItem
            sourcePath: /ClassItem[@classId='lbs#phase']/Collection[@name='Entity']
      constraints:
        - constraint:
            joinKind: inner
            collection1: WorkItemPhase
            collection2: Phase
            fieldName1: IdPhase
            fieldName2: IdPhase
      filters:
        objectItemFilter: |
          Id=#Id
```

Listing 45.3: Join collection pattern

This pattern is powerful because it keeps navigation declarative. The view consumes `WorkItemPhases`; it does not need to manually query the relation table, resolve phase objects, and assemble display rows.

45.4 Role Paths

A `rolePath` connects a projected field to a conceptual role. It is the bridge between tabular persistence and conceptual memory.

```yaml
fields:
  - IdWorkItem:
      rolePath: Id
  - ClassIdWorkItem:
      rolePath: ClassId
  - IdPhase:
      rolePath: Phase.Id
  - FormalNamePhase:
      rolePath: Phase.FormalName
```

Listing 45.4: Role path examples

A role path should be read as a semantic path, not only as a field-mapping instruction. It tells the runtime where the value belongs in the conceptual object.

45.5 Queryable Models and Complexity

As applications grow, joins become essential. Without explicit join models, query logic tends to be duplicated inside screens, scripts, and reports. With explicit collections, the query becomes part of the reusable conceptual model.

> A collection is a named path through the conceptual memory. A join is a named projection over several paths.

The design rule is to move stable navigation patterns into model declarations. Scripts should orchestrate exceptional behavior, not duplicate structural queries that belong to the model.

Chapter 46

Application Configuration and Package Assembly

A logiCells application is assembled declaratively. The application configuration connects the conceptual runtime, graph or hypergraph storage, database aliases, packages, binary extensions, services, resources, logs, and desktops.

This makes the application boundary visible. Instead of hiding runtime assembly in code, logiCells exposes it as a configuration model.

46.1 Runtime Launch Configuration: `.lgc` YAML

A Runtime launch file with the `.lgc` extension is YAML. The `kernel` mapping describes the Runtime launch environment and the `wallets` mapping supplies wallet settings. It is distinct from the application descriptor selected by `kernel.config`.

```
kernel:
  config: 'C:\logiCells\config\BaseConsole.Memory.application.yaml'
  modules: 'C:\logiCells\modules'
  base: 'C:\logiCells\runtime'
  libraries: 'C:\logiCells\runtime\bin'
  abi: 'C:\logiCells\abi'
  dsl: 'functionalP'
  services: 'httpServ;liveness;readiness;dbgclideamon;openapiService;mcpService'
  engine: 'C:\logiCells\runtime\bin\clgc.exe'
  type: 'Application'
wallets:
  user: 'example-user'
  password: 'REPLACE_ME'
```

Listing 46.1: Runtime launch configuration (.lgc)

The paths above are illustrative deployment paths. The `kernel.config` field selects the `*.application.yaml` descriptor that assembles packages, models and services. The `kernel.abi` field points to the ABI information supplied to the engine. The `kernel.services` value is a semicolon-delimited selection of service identifiers; `kernel.dsl` selects `functionalP` in this example. The `wallets.password` value is a placeholder: credentials must be secured for the target environment.

The Runtime resolves this launch configuration before loading the application descriptor, package paths and model manifests named `_.manifest.yaml`.

46.2 Application Identity and Desktop

An application configuration begins by naming the application and declaring its desktop environments.

```yaml
configuration:
  application:
    name: BusinessApplication
    desktopName: DefaultBusinessDesktop
    desktops:
      - DefaultBusinessDesktop:
          caption: Default Business Desktop
```

Listing 46.2: Application identity pattern

The desktop is not only a window. It is the root visual context from which views, controllers, page models, and object-behind components are loaded.

46.3 Graph and Metadata Loading

The graph or hypergraph section declares how conceptual memory is hosted.

```yaml
hypergraph:
  metaDataHypergraphKind: MemoryHypergraph
  loadApplicationMetaData: false
  inference:
    mode: auto
    execution: multithread
```

Listing 46.3: Graph and inference pattern

The exact graph implementation can vary between development, test, and production. What remains stable is the role of the graph: it is the memory substrate where concepts, relations, instances, and metadata can be interpreted.

46.4 Package Items, Schema Paths, View Paths and Desktops

A package `item` is a logical activatable unit. It is not merely a folder name. It groups a coherent model/package contribution that the application can enable or ignore.

```yaml
# Business package
package:
  files:
    schemaDirectories:
      schemaPath:
        value: '[Modules]\\task-management\\meta\\models\\'

# Companion VIEW package
package:
  dependencies:
    - task-management
  files:
    schemaDirectories:
      schemaPath:
        value: '[Modules]\\task-management\\meta\\views\\models\\'
      schemaPath:
        value: '[Modules]\\task-management\\meta\\views\\contracts\\'
  items:
    - TaskManagement.Task:
        caption: Task
        enabled: true
```

Listing 46.4: Business package and companion VIEW package roots

`schemaPath` tells the package loader where manifests and model-bearing structures are located. The companion VIEW package uses two roots with distinct responsibilities: `views/models` for MODEL redefinitions/overlays and `views/contracts` for interaction contracts and their colocated MODEL-behind containers. VIEW-package items are relative to these roots; they should not be prefixed with `contracts.` or `models.` solely because of physical placement.

A VIEW package must remain scoped to the interactions owned by its business package. A missing historical `viewPaths` declaration is not sufficient reason to claim the complete global `meta/views` tree.

The `desktops` section declares application desktop entry points or desktop assemblies. It is the bridge from a loaded application to a navigable desktop experience. Exact attributes depend on the desktop runtime, but the architectural role is stable: it selects the root interface configuration rather than defining business concepts.

46.5 Database Aliases

A database alias binds a logical database name to a concrete driver configuration.

```
dataBaseAlias:
  - dBAlias:
      name: Default
      items:
        - lgc#default_group:
            type: SQLite
            driverPackage: logiCells.Data.FireDac
            value: DriverID=SQLite;database=[BasePath]/database/application.db3
```

Listing 46.5: Database alias pattern

Database details should not leak into the conceptual model. The model declares facets and maps; the application configuration chooses the runtime database binding.

46.6 Binary Packages and Model Packages

A logiCells application is assembled from several kinds of packages.

```
container:
  binaryPackages:
    - logiCells.Community
    - logiCells.SQLDataStructure
    - logiCells.Agent
    - logiCells.MVCCore
  packages:
    - package:
        path: '[Modules]/logicells-kernel-open/packages/Common.package'
        caption: Kernel
        enabled: true
        groupId: lgc#default_group
    - package:
        path: '[Modules]/business-application/packages/BusinessApplication.package'
        caption: Business Application
        enabled: true
        groupId: lgc#default_group
```

Listing 46.6: Runtime package assembly pattern

This pattern separates reusable platform packages from application packages. A platform package may provide entities, messaging, views, controllers, or services. An application package contributes domain models, process models, application views, and specific resources.

46.7 Platform and Application Namespaces

Public examples may use both platform and application namespaces. The platform namespace generally uses `lgc#`. Application examples may use a namespace such as `lbs#`. This separation is useful because it makes ownership between platform concepts and application concepts visible.

```
lgc#entity
lgc#process
lgc#custom_message
lbs#work_item
lbs#work_item_process
lbs#phase
```

Listing 46.7: Platform and application identifiers

The rule is not that every identifier must be `lgc#`. The rule is that platform concepts and application concepts must be clearly separated.

46.8 Services and Files

The same configuration can declare service definitions, temporary folders, logs, caches, synchronization repositories, snapshots, workflow repositories, and wallets.

```
files:
  tempDirectory: '[BasePath]/temp/'
  logDirectory: '[BasePath]/log/'
  cachePath: '[BasePath]/cache/'
  synchronizationRepository: '[BasePath]/synchronization/'
  snapshotRepository: '[BasePath]/snapshots/'
  workflowBaseRepository: '[BasePath]/process/'
```

Listing 46.8: Runtime file locations pattern

These locations are operational details, but they are still part of the declarative application boundary.

46.9 Reusable Platform Capabilities

Application assembly is also where reusable platform capabilities become visible. A logiCells application is not only a set of domain concepts. It is assembled from platform packages that may contribute messaging, cache management, visual components, services, workflows, synchronization, agents, resources, styles and runtime providers.

```
Platform packages
  -> kernel concepts
  -> MVC and cache layer
  -> UXfc functional components
  -> messaging and conversations
  -> services and API adapters
  -> synchronization and snapshots
  -> agents and workflow execution

Application packages
  -> domain concepts
  -> application processes
  -> views and page models
  -> providers and integrations
  -> resources and styles
```

Listing 46.9: Reusable capabilities in a package assembly

This packaging discipline is important because it separates what belongs to the platform from what belongs to a specific application. The same messaging platform, visual functional components, data cache layer, provider architecture or service adapter pattern can be reused in several applications without exposing client-specific workflows or visual details.

46.10 Local and Remote Runtime Resources

An application configuration may also declare operational repositories used by local-first or distributed deployments.

```
cache
  view-ready data and provider results

synchronization
  pending changes and remote replication state

snapshots
  coherent saved states or exchange packages

process
  workflow and process execution state

resources
  icons, styles, layouts and static assets
```

Listing 46.10: Runtime resource categories

These directories are not merely file system details. They correspond to runtime responsibilities: local cache, synchronization, packaging, recoverability, process execution and user experience resources.

46.11 From Package to Loaded Model

A file present under `meta/models` is not loaded merely because it exists on disk. A new module becomes part of an application through an explicit loading chain:

```
application configuration
  -> package descriptor
  -> schemaDirectories / schemaPath
  -> _.manifest.yaml
  -> declared model
  -> runtime ClassItem
  -> semantic initialization when applicable
```

Listing 46.11: Declarative module loading chain

When creating a module, verify this chain from the outside inward. This is especially important when two checkouts contain similarly named models or when a historical package still exposes an older declaration.

A recommended public project shape for a self-contained module is:

```
my-module/
  packages/
    MyModule.package.yaml
    MyModule.Tests.package.yaml
  meta/
    models/
      MyModule/
        _.manifest.yaml
        MyModel.model.yaml
      MyModuleTests/
        _.manifest.yaml
        MyModelTests.model.yaml
  src/
  tests/
```

Listing 46.12: Recommended module shape

The exact set of source files depends on the module, but the package and manifest remain the declarative entry points.

46.12 Compiling and Starting an Application from the Command Line

On runtime distributions that provide the `sclgc` host, it can load and start an application from a local `.lgc` configuration with `-c`:

```
sclgc -c /path/to/application.application.ACC.lgc
```

Listing 46.13: Starting a configured application with sclgc

The `-c` argument identifies the local configuration to load. The configuration then resolves the application, module paths, package assembly and other runtime parameters. Run the executable from an environment where its runtime binary packages and dependent libraries can be resolved; in a development tree this is often the runtime `bin` directory used by the build.

This command is different from the design-time Tooling CLI. The Tooling CLI exposes application compilation through the native application loader, while `sclgc -c ...` starts the runtime using the selected local configuration. Do not treat the two command surfaces as interchangeable.

Local configuration values may also depend on process environment values. When the Tooling command surface accepts an explicit `-param`, the validated precedence is:

```
explicit --param
  > value in .lgc
  > process environment
  > unresolved
```

Listing 46.14: Configuration value precedence

The current application parameter authority accepts both a long form with a double dash and a short form with a single dash. For example, `-config path` and `-c path` resolve to the same application parameter. Parameters declared without a value are flags; parameters declared with a value consume the following command-line argument.

Long name	Short	Value	Purpose
`config`	`c`	yes	Path to the configuration file.
`roles`	`r`	yes	User roles or access levels.
`debug`	`dbg`	no	Enable detailed debug mode.
`listeners`	`ls`	yes	Number of server listeners.
`send-to`	`sto`	yes	Destination for data transfer.
`services`	`sg`	yes	Services initialized at startup.
`base`	`b`	yes	Base directory.
`modules`	`m`	yes	Modules to load.
`libraries`	`l`	yes	Additional libraries.
`language`	`lg`	yes	Application language.
`test`	`t`	no	Run in test mode.
`test-out`	`to`	yes	Test result output directory.
`test-application`	`ta`	no	Enable application tests.
`test-hypergraph`	`th`	no	Enable the test-hypergraph option in the current parameter contract.
`test-hypergraph-params`	`thp`	no	Enable the test-hypergraph-params option in the current parameter contract.
`script`	`f`	yes	Script file to run.
`db-name`	`d`	yes	Database name.
`db-user`	`dbu`	yes	Database user.
`db-password`	`dpb`	yes	Database password.
`server`	`s`	no	Run the server instance.
`log-level`	`ll`	yes	Logging level.
`configuration`	`a`	no	Enable alternative configuration settings.
`ast-debug`	`astsd`	no	Enable script/AST debugging.
`memory-profiler`	`mp`	no	Enable memory profiling.

`cache-view`	`cv`	no	Display cache view.
`dsl`	`dsl`	yes	DSL file to load.
`minimal`	`mi`	yes	Minimal-mode parameter.
`style`	`st`	yes	Style configuration.

A transient debugging build may additionally expose `-followed-index`/`-fi` with a value. Because it is conditional on the `TRANSIENTMLDEBUG` build define, application documentation should not rely on it as a portable runtime option.

The parser performs strict validation when strict parameter control is enabled: unknown options are rejected, and an option that requires a value is rejected when it is the final argument without one. Long options use `-name`; short options use `-name`. This parameter surface belongs to the application runtime and should not be inferred from another CLI facade.

46.12.1 What Success Looks Like

For a declarative module change, a useful smoke test is not only process startup. Follow the loading stages in the log:

```
Loading application file
Loading package ...
Compiling package item ... _.manifest.yaml...
Compiling model ...
Start initializing/updating semantic Hypergraphs
End initializing/updating semantic Hypergraphs
```

Listing 46.15: Runtime smoke-test checkpoints

If the model does not appear in this sequence, diagnose package and manifest assembly before investigating actions, prompts or business behavior.

46.13 Design Rule

The application configuration answers the question:

> Which runtime, packages, graphs, databases, services, resources, and desktops are assembled to make this conceptual application executable?

Keeping this answer declarative makes environments easier to reproduce, inspect, deploy, and evolve.

46.14 Memory and Persistent Starter Configurations

The starter project contains production-like application configuration files instead of minimal placeholders. The starter uses YAML for application configuration; its visual sources remain canonical `*.view.xml`.

```
yaml/configurations/BaseConsole.Memory.application.yaml
yaml/configurations/BaseConsole.Persistent.application.yaml
```

Listing 46.16: Starter application configurations

The memory configuration is the recommended first run. It lets the developer check metadata loading, package assembly and model visibility before dealing with persistent storage. The persistent configuration is closer to a real environment, but it requires explicit adaptation of database aliases, local paths, credentials, ports and service activation.

A good debugging order is:

```
1. Start with the Memory configuration.
2. Verify that the package is enabled.
3. Verify that the manifest exposes the expected class.
4. Verify that ObjectsContainer can resolve the class.
5. Open the view and check DataSource bindings.
6. Switch to the Persistent configuration.
7. Check database aliases and maps.
8. Activate services only after the model works.
```

Listing 46.17: Configuration debugging order

This order prevents a common beginner mistake: debugging services or database writes while the package or manifest is not loaded correctly.

46.15 Compiled Metadata Versus Semantic Graph Initialization

An application configuration selects source packages, source locations, metadata baselines, runtime groups and related services. In the native-strict cold pipeline, `TMetadataNativeStore` backs assembled metadata and emits `.lcmp`; the old metadata `MetaDataHypergraph` does not remain a hidden fallback. The semantic `HypergraphBundle` is an independent and valid target for H-Logic and logical relations. Distinguish the source digest/compile plan from the compiled generation, semantic graph generation and SQL persistence generation. The configuration's compiled-only warm branch must validate both source and security rather than silently force a facade binding.

The runtime provides optional compilation and bootstrap performance switches; they do not change the application model. Do not set `LGC_INCREMENTAL_SIGNATURE_CACHE_REV` without proving the pinned YAML parser/DOM/grammar revision; omission simply restores normalized signature calculation. The relevant developer workflow is in Chapter 47.

Chapter 47

Incremental Compilation, Impact Analysis and Native Metadata

47.1 Understand the Authority Before Selecting a Command

The `cell` command is a client of native logiCells Tooling, not another metadata engine. Choose the operation according to its authority:

MODEL Inspect or edit physical declarative XML/YAML through the native parser and source DOM. A MODEL operation does not implicitly load an application or semantic graph.

APPLICATION Inspect or mutate assembled metadata for a resolved `.lgc` configuration, including native facades. An APPLICATION mutation can publish a new durable metadata generation even when no physical source file is available.

Compilation plan Read the source-delta plan; do not mutate the application.

Compile Apply an incremental delta only when it is safe. A refused incremental compile does not silently become a full build.

Build Explicit full build, with a new baseline after successful completion.

47.2 Practical Developer Workflow

```
cell doctor
cell workspace inspect <workspace>
cell application inspect <application.lgc>
cell application compile plan <application.lgc>
cell application compile <application.lgc>
# if the plan requires a full rebuild, decide explicitly:
cell application build <application.lgc>
```

The plan separates `ChangedUnits`, directly `DirtyClasses`, the `AffectedClasses` closure, typed propagation reasons, eligibility, `fullRebuildRequired` and `fallbackReason`. Do not infer that `compiled=True` entails an applied change: an unchanged source may be compiled successfully while `applied=False`. The package-level reference commands are `cell package compile` and `cell package build`. When `package compile --verbose` succeeds, its reported plan comes from the execution response, not necessarily from an independent pre-execution plan call.

The source-driven planner primarily accepts modified class units in the characterized implementation; additions/deletions of other kinds can remain unsupported. A plan with `changed=0`,

`dirty=0` and `affected=0` establishes only that the current baseline was considered unchanged by the tested signature algorithm.

47.3 What Changed: Source, Aspect and Dependency

Incremental compilation compares a normalized DOM-based digest and six signatures (structure, semantics, behavior, state, persistence and view). Typed dependency edges then determine which consumers are affected; unrelated model neighbors are not automatically dependents. A change to whitespace or comments may modify physical file bytes while leaving normalized metadata unchanged. A dependent Action may require recompilation without a change in its own source when the types or role projections it uses change.

```
source file -> native XML/YAML parser -> normalized digest
                                      -> six aspect signatures
                                      -> typed dependency closure
                                      -> eligible incremental plan
```

The `1/1/7` result in the reference `lgc#folder` fixture is not a general rule. Distinguish the `ClassChangeIsFieldAddOnly` optimization from ordinary dependency propagation. If the typed closure or available signatures are uncertain, the compiler must preserve conservative fallback behavior.

47.4 NativeStore Does Not Replace the Semantic Graph

The assembled `TXXXInfo` metadata model is backed by `TMetadataNativeStore` and compiled `.lcmp` facades on native-strict paths. The historical metadata backing `MetaDataHypergraph`/`FDataContent` is not an allowed fallback in the native-strict cold runtime. The `HypergraphBundle` semantic graph remains a separate authority for reasoning, H-Logic, operationalization and context. Thus a successful metadata compilation does not automatically prove that semantic `.sem` artifacts, graph types or SQL schema are current.

47.5 Cold, Warm and Physical YAML Cache

The optional setting `LGC_COLD_MODEL_SINGLE_DOM=1` avoids constructing a second parsed DOM for eligible cold YAML model loads. Another optional optimization uses the `.units.physical-signatures` sidecar to reuse unchanged normalized digests and aspect signatures. The latter requires a nonempty, deployment-maintained `LGC_INCREMENTAL_SIGNATURE_CACHE_REV` and a complete, committed baseline with matching source identity, physical MD5 and parser compatibility; any uncertainty returns to historical parser/DOM computation. The cache is not a source of business truth, and does not replace `typed-v1` dependency analysis.

Source-signature caching can reduce repeated parsing of unchanged units, but results depend on the project, runtime profile and source set. The `.lcmp` image cache and the physical-signature sidecar have different purposes: one reuses compiled metadata, while the other avoids redundant signature computation. Validate artifact equivalence, invalidation after source changes and concurrent build behavior on your own workload.

47.6 Inspecting a Failure Without Weakening Invariants

If a cold execution reaches `functionalP>`, its minimal startup worked; inspect unresolved `#supertype` diagnostics, type-shape compatibility, generated artifacts and any changed fixture before claiming full semantic closure. If warm compiled loading rejects native binding, investigate signature

provenance and security bootstrap. A trace such as `applicationMD5=` followed by `native-store-reparse` describes a fallback to diagnose, not permission to force binding or invent a signature. The observed v37 warm attempt remained open.

47.7 Impact Analysis in VS Code: The Correct Boundary

The VS Code integration may present changed and affected units alongside source locations. Consult the installed extension for the exact impact-view capabilities; do not infer compiler safety from an editor-side approximation.

47.8 Developer Verification Checklist

1. Capture the source and compiler revisions, the full selected command and Tooling response.
2. Verify `ChangedAspects` and typed reasons rather than trusting the closure size alone.
3. Run a deliberately changed model and its immediate `0/0/0` rerun; test restoration as well as addition.
4. Check stable IDs and tombstones for collections and their structural children.
5. Compare relevant `.lcmp`, `.ast` and `.sem` outputs on controlled cache ON/OFF tests.
6. Separate tested cold startup from semantic correctness, artifact parity, concurrent publication and warm security binding.

The native algorithms and the dated proof matrix are described in the companion Runtime RadStudio book, chapters on NativeStore and incremental metadata compilation. This chapter documents the developer-facing interpretation rather than defining an alternative compiler.

Chapter 48

Declarative Runtime Services and Engine Patterns

A service declaration does not implement a scheduler, queue, agent or journal by itself. It selects, configures and instantiates a runtime implementation registered by the engine or by a binary package. Understanding this bridge is essential when reading production service files.

48.1 From Declaration to Runtime Behavior

```
service metadata
-> parser and declarative structure
-> service definition
-> moduleClass / runtime registry lookup
-> configured runtime instance
-> start/stop lifecycle
-> events, timers, queues or HTTP behavior
```

Listing 48.1: Declarative service resolution chain

The declaration is therefore an executable configuration contract. It names a runtime family, provides parameters and determines which instances are active.

48.2 Three Service Levels

Element	Role	Typical content
`serviceGroup`	Package-level publication and deployment grouping	definition path and one or more instances
`serviceGroupDefinition`	Reusable service type/configuration contract	`uuid`, `name`, `type`, `moduleClass`, pages, parsers, parameters
`serviceInstance`	Concrete running instance	address, port, database, configuration, active/start settings

48.3 Common Definition Attributes

Attribute	Meaning
`uuid`	Stable identity of the definition. Keep it stable when deployments or references depend on it.

`type`	Broad service family, such as HTTP or another engine-supported service category.
`moduleClass`	Runtime class or module responsible for implementing the declared behavior. This is the critical bridge from metadata to code.
`debug`	Enables service-level diagnostics where supported. Do not rely on it as the only logging control.
`start`	Indicates whether or how the service participates in startup, depending on the runtime generation. Validate the accepted values against the target engine.
`baseURL`	Base address used to compose or publish routes.
`path`	Current path or routing location.
`path_old`	Historical compatibility path seen in older projects. Do not introduce it in new code unless migration behavior requires it.
`defaultPages`	Default pages or handlers used when a request does not select a more specific route.

48.4 Additional Sections

`views` Declares view or page resources exposed by the service boundary.

`parameters` Supplies typed or named configuration values to the runtime module.

`viewParsers` Registers parsers or transformations used to interpret published views or payloads.

`binaryPackages` Loads runtime implementations required by the service definition.

These sections should be documented from the runtime class that consumes them. A similarly named section in another service family may have different guarantees.

48.5 Built-In Architectural Patterns

Production configurations may contain names such as agents, queues, journals, operations, rules or timers. Treat these names as runtime service families, not as proof that every textbook property of the corresponding design pattern is implemented.

Family	**Typical intention**	**Questions to verify**
Agent	Autonomous or delegated execution	trigger, permissions, state, retry and action boundary
Queue	Buffered asynchronous work	delivery guarantee, ordering, persistence, retry and dead-letter behavior
Journal	Durable trace or event record	retention, ordering, replay and transaction boundary
Operation	Named executable service operation	input contract, target, result and failure semantics
Rules	Declarative evaluation or dispatch	evaluation context, ordering, conflict behavior and side effects
SimpleTimer	Repeated interval-based trigger	interval, startup, overlap and failure behavior

calendarTimer	Calendar/schedule-based trigger	timezone, missed runs, recurrence and concurrency

The guide intentionally avoids claiming guarantees that are not confirmed by the corresponding runtime implementation and tests.

48.6 Example: HTTP Service Definition

```
serviceGroupDefinitions:
  - UserManagement:
      uuid: EA588812932A434B87A5F47A795C27E4
      type: HTTP
      moduleClass: DataModuleHTTPServer
      debug: true
      virtualPages:
        - method: GET
          path: /users
          class: PathRequestWebPage
          rewrite: /ClassItem[@name='UserManagement.User']/Collection[@name='Entity']
```

Listing 48.2: HTTP service definition in YAML

48.7 Virtual HTTP Pages and Page-Extension Files

The HTTP service declaration registers virtual routes under `virtualPages`. In the example above, an active HTTP service receiving `GET /users` can select the rule with `method: GET` and `path: /users`. The declared `class` is the ABI-facing handler `PathRequestWebPage`; `rewrite` identifies the target conceptual collection. A different method or path does not select this particular route.

HTTP page-extension files use the `.lgp` suffix, for example `Users.lgp`. The page file and the `virtualPages` route are distinct resources: a file is not a published route merely because it exists. The example does not specify a physical page-file loader field; the mapping between a page resource and the registered handler must follow the actual target Runtime contract.

```
HTTP service running
-> GET /users
-> matching virtualPages entry
-> PathRequestWebPage (ABI handler)
-> rewrite to UserManagement.User / Entity collection
-> HTTP service response
```

Listing 48.3: Declarative request selection for the HTTP virtual-page example

Request parsing, authorization, handler invocation and physical `.lgp` resolution are implementation boundaries. The exact runtime callback cannot be determined from declarative metadata alone.

48.8 Adding a New Runtime Pattern

Adding a new declarative service family generally requires more than adding a new declarative metadata rule:

1. implement or reuse a runtime class;
2. define the metadata contract it accepts;

3. register the implementation in the engine or a binary package;
4. expose configuration properties and lifecycle hooks;
5. declare an instance in service metadata;
6. test startup, shutdown, error behavior, events and persistence;
7. document the guarantees that callers may rely on.

This extension boundary is one of the strengths of logiCells: new runtime capabilities can become declaratively composable once their metadata contract and registration are stable.

48.9 Reference Strategy

An exhaustive inventory of every historical service tag would quickly become stale. The recommended documentation pattern is:

```
conceptual chapter
  explains the declaration-to-runtime mechanism

runtime family reference
  documents accepted attributes and lifecycle guarantees

validated examples
  show configurations tested with a named engine generation

source/tests
  remain the authority for implementation-specific behavior
```

Listing 48.4: Service documentation layers

Chapter 49

Deployment, Packaging, and Runtime Distribution

Once an application has been modeled, it must be packaged and distributed. logiCells deployment is based on the same principle as modeling: make the structure explicit.

A deployment package is not only a compiled program. It is an application tree containing runtime binaries, model packages, views, services, resources, databases, configuration files, logs, caches, synchronization folders, and update descriptors.

49.1 Supported Runtime Platforms

logiCells is currently available for **Windows**, **Linux** and **macOS**. The runtime can also be deployed in a **container**.

The executable conceptual model is not tied to one of these operating systems. Platform-specific packaging, native libraries and deployment settings are operational concerns around the same model and runtime contracts.

A container packages a runtime for deployment; it does not define a different application model.

49.2 Platform Availability versus Subsystem Certification

The runtime distribution is available on all three desktop/server operating-system families above. That statement must not be widened into a claim that every native integration path has the same certification status on every host. The installation and development toolchains differ by host, and availability of the main runtime does not guarantee that a particular native-module loader is supported on every platform. Check the host-specific ABI and binding requirements of the selected distribution.

For adoption work, record platform status per subsystem rather than with one global checkbox:

Surface	Current status	What the statement means
Core runtime distribution	Windows / Linux / macOS	The conceptual runtime is available on these platforms.
Container deployment	Available	A runtime can be packaged and operated in a container; image and host details remain deployment-specific.

Native development/build baselines	Documented on Windows / Linux / macOS	Choose the matching build tools and runtime package for each intended host.
Native ABI dynamic loading	Platform-specific certification required	Do not infer Linux/macOS native-loader certification from runtime availability; verify the binding package for the selected host.
Python / C# ABI bindings	Available	Binding availability and lifecycle coverage are documented in Chapter 64, “Declarative Models and the Runtime ABI”; host-specific packaging must still be verified.

This distinction is intentional. A platform statement describes where the runtime is available; a certification statement describes a tested path through a particular subsystem, backend and packaging configuration.

49.3 The Deployment Tree

A deployment tree usually separates stable platform resources from application resources.

```
application/
  packages/
  models/
  views/
  services/
  resources/
  database/
  database/clean/
  graph/
  cache/
  log/
  temp/
  synchronization/
  snapshots/
  process/
  wallets/
```

Listing 49.1: Deployment tree pattern

This structure matters because logiCells applications are metadata-driven. Models and views are runtime assets, not only development artifacts.

49.4 Packaging Platform and Application Assets

A static generation or packaging configuration commonly copies several families of files:

- kernel and platform packages;
- model files;
- view files;
- service definitions;
- resources and styles;
- database templates or clean databases;

- grammars and scripting assets;
- application configuration files.

```
finalTreeInfos:
  - finalTreeInfo:
      targetPlugIn: common
      name: Models_kernel
      sourceFile: '#models_kernel#*.*'
      targetDirectory: '#target_data#models/'
  - finalTreeInfo:
      targetPlugIn: application
      name: Models_application
      sourceFile: '#models_application#*.*'
      targetDirectory: '#target_data#models/'
  - finalTreeInfo:
      targetPlugIn: application
      name: Views_application
      sourceFile: '#views_application#*.*'
      targetDirectory: '#target_data#views/'
```

Listing 49.2: Abstract static generation pattern

The names of concrete directories vary by project, but the pattern is stable: platform assets and application assets are assembled into a runtime tree.

49.5 Configuration by Environment

Development and release configurations may differ. Development may use memory graphs, local paths, debug logs, and test databases. Release may use persistent graphs, packaged resources, clean database templates, and fixed runtime folders.

```
# Development
(graphKind: MemoryRichGraph, graphParams: [Workspace]/Graph)

# Release
(graphKind: PersistentRichGraph, graphParams: [BasePath]/Graph)
```

Listing 49.3: Environment-specific graph binding

The conceptual model should remain stable across these environments. The configuration binds it to the concrete runtime deployment.

49.6 Updates and Runtime Evolution

An update descriptor can declare modules, versions, and download locations. This keeps deployment evolution explicit.

Update mechanisms are operational, but they participate in model governance. If models, views, and services are runtime assets, updating them is part of application evolution.

49.7 Containerized Deployment

A logiCells application can be packaged and run in a container. For service-oriented deployments, application folders can be mounted as volumes and the runtime can expose the required service ports.

```
services:
  business-application:
    image: business-application
    ports:
```

```
5       - 9696:9696
6     volumes:
7       - ./deploy/database:/var/application/database
8       - ./deploy/temp:/var/application/temp
```

Listing 49.4: Container deployment pattern

The important point is that containerization does not replace the conceptual runtime. It packages it.

49.8 Design Rule

A deployment should make the following boundaries visible:

- what belongs to the platform;
- what belongs to the application;
- what is configuration;
- what is generated;
- what is persistent runtime state;
- what can be updated independently.

Deployment is the operational continuation of declarative modeling.

Chapter 50

Blueprint for a New ERP Module

This chapter is the default starting point for a new ERP capability. It combines the source conventions, MODEL boundaries, VIEW contract, tests and package assembly into one repeatable slice.

50.1 1. Start from a bounded business capability

Name the module after a stable business capability, not after a screen. Examples include `erp-procurement`, `erp-scheduling`, `erp-training` or `erp-expense`. Write the invariants and lifecycle before creating views.

50.2 2. Recommended module tree

```
Modules/erp-example/
  meta/
    models/
      _.manifest.yaml
      BusinessObject.model.yaml
      BusinessRelation.model.yaml
      BusinessProcess.model.yaml
    views/
      BusinessObject/
        _.manifest.yaml
        BusinessObjectList.view.xml
        BusinessObjectList.models.yaml
        BusinessObjectEdit.view.xml
        BusinessObjectEdit.models.yaml
  packages/
    erp-example.package.yaml
    erp-example.view.package.yaml
Tests/
  twin-test-example/
    ...
```

Listing 50.1: ERP module skeleton

50.3 3. Model identity and invariants first

Use stable `classId` values. Keep identity and required relationships explicit. Put validation next to the business model rather than relying on a form to make a field required.

```
class:
  name: BusinessObject
  classId: erp#example_business_object
  currentGeneratorName: Uri
```

```yaml
type: Entity
concepts:
  - Entity:
      facets:
        - main:
            type: hypergraph
            fields:
              - Number:
                  type: String
                  nullable: false
              - Status:
                  type: String
                  nullable: false
                  defaultValue: DRAFT
      actions:
        - action:
            name: Validate
            class: ObjectItemScriptableObjectElementAction
            type: ObjectItem
            published: false
            scripts:
              - script:
                  clientScript: false
                  serverScript: true
                  language: functionalP
                  cdata: |
                    Result := '';
                    if Self.Number = '' then
                      Result := 'Number is required';
```

Listing 50.2: ERP entity skeleton

50.4 4. Actions express reusable business intentions

An action should be callable from a view, service, test or process without duplicating its invariant. Favor idempotent operations when retries are plausible. Separate destructive cleanup from business deletion when temporary/snapshot objects exist.

50.5 5. Collections are reusable read models

Do not build every screen query inside the VIEW. Declare collections/projections once, name them by business meaning, and reuse them from dashboards, services and tests. If a historical report is being migrated, recover its semantic projection instead of recreating a navigation-root object merely because the legacy application had one.

50.6 6. Processes coordinate objects

Use a process when a business transition coordinates several objects or steps. VIEW scripts should initiate or observe the process, not implement its transaction choreography.

50.7 7. Add the VIEW package after the business contract

Create `*.view.xml` for the visual hierarchy and `*.models.yaml` for controllers. Keep controllers client-oriented and thin. Use explicit visual events and lifecycle continuations for asynchronous interactions.

50.8 8. Test the contract in layers

A new ERP slice should have at least:

1. MODEL parse, manifest and class-identity checks;
2. action contract tests for validation and state transitions;
3. collection and read-model contract tests;
4. process tests for cross-object orchestration;
5. VIEW binding tests: each XML `source/class/classId` resolves exactly once;
6. asynchronous VIEW protocol tests for waiting, completion and refresh;
7. package and application assembly checks;
8. runtime tests in the supported logiCells environment before release.

50.9 9. Definition of done

A module is not complete because its files parse. Before calling an ERP capability complete, establish:

- stable business identity and invariants;
- reusable actions and collections;
- explicit process ownership for cross-object flows;
- VIEW/model-behind separation;
- bounded event-driven asynchronous behavior;
- package and manifest completeness;
- tests that prove the business contract;
- runtime validation or an explicit statement that runtime validation is still pending.

Part VII

Runtime Semantics

Chapter 51

From Declarations to Runtime Execution

A logiCells application begins as a set of declarations, but it does not remain a passive description. A `.model.yaml` file is loaded into the conceptual runtime, turned into metadata, attached to a graph, and made executable through runtime objects. The important transition is therefore not "model to code". It is "declaration to executable concept".

This chapter explains that transition. It shows how a declarative model becomes a runtime structure made of class items, object items, actions, collections, events, views and services.

51.1 The Runtime Path

The execution path can be read as a chain of responsibilities:

```
.model.yaml
  -> metadata graph
  -> ClassItem
  -> ObjectItem
  -> facets and fields
  -> collections and providers
  -> actions
  -> events
  -> views and services
```

Listing 51.1: From declaration to executable runtime structure

The declaration defines what a concept is. The runtime gives that concept an executable form.

A `ClassItem` represents the concept as a runtime class-like object. It knows its class identifier, declared facets, collections, actions, views and generators. An `ObjectItem` represents an instance of that concept. It carries values, participates in relations, exposes actions and can be persisted through the declared facets.

The runtime container is the environment where these objects are resolved and executed. It provides operations such as finding a class by its class identifier, creating a new object item, locating an existing object item, opening a collection provider, dispatching an event or persisting changes.

51.2 Class Items and Object Items

A class declaration such as `lgc#custom_message`, `lgc#conversation` or `lbs#work_item` becomes a runtime class item. The class item is not merely a schema. It is the runtime carrier for:

- the concept identity;
- the declared facets;
- the available collections;
- class-level and object-level actions;
- event subscriptions;
- view declarations;
- generators and default values.

An object item is then created from that class item. It is the executable conceptual instance. For example, a work item can be created, attached to a customer, persisted, displayed in a view, used by a workflow and emitted in an event.

51.3 Runtime Containers

The runtime container is the operational boundary of the application. It holds access to metadata, object instances, collections, action dispatching, event routing and services. In scripts, this role is often visible through expressions such as:

```
1 WorkItemClass := Self.ObjectsContainer.ClassItemByClassId('lbs#work_item');
2 WorkItem := Self.ObjectsContainer.CreateObjectItem(WorkItemClass, 1, '');
3 WorkItem.Name := 'Initial assessment';
4 WorkItem.PostData();
```

Listing 51.2: Typical runtime container operations

The example shows a simple runtime pattern:

1. resolve the concept through its class identifier;
2. create an executable object item;
3. assign conceptual values;
4. persist the object through its declared persistence facet.

The same pattern appears in process coordination, messaging, view controllers and service adapters. logiCells applications are therefore not organized around isolated code classes. They are organized around runtime concepts made executable by the container.

51.4 Facets at Runtime

Facets explain how a concept is projected into different technical and conceptual spaces. A hypergraph facet describes the conceptual roles and relationships. A database facet describes how part of the same conceptual structure is persisted. A view or service facet may project the same concept toward an interaction or an integration boundary.

The runtime does not treat those facets as unrelated schemas. It uses them as coordinated projections of the same concept. This is why a field may have a `rolePath` such as `Customer.Id` or `Conversation.ClassId`: the persisted representation stores a projection of a conceptual relation.

51.5 Actions as Runtime Entry Points

Actions are the most visible entry points into runtime execution. A declaration may expose an object-level action, a class-level action, a scriptable action or an action implemented by a registered runtime component. The key point is that the action belongs to the concept.

The following sketch shows the pattern:

The script is an implementation detail. The action itself is the declared business capability.

51.6 Events and Reactive Execution

At runtime, an event connects a source, a payload and one or more actions. Events allow the platform to avoid hidden coupling between business objects, process objects and views. A process can emit an event; a controller can subscribe to it; a view can react to it; a messaging component can propagate it asynchronously.

This is the main difference between procedural orchestration and conceptual runtime execution. In a procedural system, the caller must know the callee. In a conceptual runtime, a concept emits a meaningful event and the runtime routes that event to the declared actions.

Chapter 52

Conceptual Memory and Change Propagation

Conceptual memory is the runtime space where concepts, objects, relations, actions and events become connected. It is not a passive storage layer. It is the place where changes can be interpreted, propagated and observed.

52.1 Memory as a Conceptual Graph

When a model is loaded, its declarations become runtime metadata. Class items define executable concept types. Object items instantiate those concepts. Relations connect object items. Events and actions give the graph behavior.

A change in conceptual memory may come from a user action, a webservice call, an agent, a script, a message, a synchronization event, or a rule evaluation. The runtime should preserve the meaning of the change: which concept changed, which role changed, which process is affected, and which observers may react.

52.2 From Local Change to Propagation

A local change usually follows a pattern:

1. an object role or field is modified;
2. the object is validated in its current context;
3. the change is posted;
4. dependent collections, views or processes are refreshed;
5. events may be emitted to interested participants;
6. remote synchronization or messaging may publish the change.

```
1 WorkItem.Status := 'Validated';
2 WorkItem.PostData();
3 Event := Self.ObjectsContainer.BuildMessage('lbs.WorkItem.Validated');
4 Self.ObjectsContainer.SendEvent(Event, Self.EventProxy);
```

Listing 52.1: Local change followed by persistence

52.3 Observable Changes

A change becomes more robust when it is observable. A hidden assignment in a script is difficult to trace. A posted change followed by an explicit event can be traced, replayed, audited or routed to a process.

52.4 Design Rules

Keep business changes close to the concept that owns them. Emit events when other participants must react. Use process models when several objects must be coordinated. Avoid invisible cross-object side effects. Make important state changes explicit and traceable.

52.5 Summary

Conceptual memory gives the runtime a common space for objects, relations, processes, views, services and agents. Change propagation is reliable when changes are expressed as conceptual updates and observable events.

Chapter 53

Evaluation Contexts

An evaluation context defines the conditions under which a model element is interpreted. The same concept may be evaluated differently depending on the current object, current user, current process, current view, current service call or current agent.

53.1 Why Context Matters

A declarative model avoids hard-coded wiring, but it does not remove context. Instead, it makes context explicit. A collection displayed in a view may depend on the selected object. A published action may be authorized only for a role. A service call may carry a tenant or application namespace. An agent may run with a limited execution envelope.

53.2 Common Contexts

The most frequent contexts are:

- object context: the current `ObjectItem`;
- class context: the current `ClassItem`;
- view context: the current frame, cursor and dataset;
- process context: the current workflow state and process object;
- security context: the user, roles and permissions;
- publication context: the service, endpoint and protocol;
- agent context: the task, memory slice and allowed actions.

53.3 Context Propagation

A view can pass its current object to a nested cursor. A process can pass itself as the source of an event. A service can map request parameters into action parameters. An agent can receive a constrained set of objects and actions.

53.4 Design Rules

Do not rely on implicit context when a model boundary is crossed. Make context explicit when calling a service, dispatching an event, opening a view, or starting a process. Avoid global assumptions that make a model difficult to test.

53.5 Summary

Evaluation contexts allow the same conceptual model to support users, views, processes, services and agents without duplicating the model for each execution path.

Chapter 54

Transactions, Consistency and Identity

logiCells applications manipulate conceptual objects, but those objects still need identity, consistency and reliable persistence. This chapter explains how to think about identity and transaction boundaries when models are executable.

54.1 Conceptual Identity

An object is identified by more than a database key. It has a concept class, an object identity, a formal name and often a contextual role in another object. Database facets typically expose this identity through fields such as `Id`, `ClassId` and `FormalName`.

54.2 Consistency Across Facets

The hypergraph facet expresses conceptual structure. The database facet provides persistence and queryable projection. Consistency means that both facets describe the same object from different operational viewpoints.

A role such as `Owner` can be represented in memory as an object reference and in the database projection as `Owner.Id`, `Owner.ClassId` and `Owner.FormalName`. The model should make that mapping explicit.

54.3 Transaction Boundaries

A transaction boundary should follow a business intention. Creating a message, validating a work item, finalizing a process or publishing a service response are natural transaction boundaries. A large script that updates many unrelated objects without a process model is a warning sign.

54.4 Process-Level Consistency

When several objects must change together, introduce a process object. The process can store intermediate state, coordinate validation, emit events and make retries possible. This is especially important for asynchronous workflows and external integrations.

54.5 Design Rules

Use stable identifiers. Project conceptual references with role paths. Call `PostData` only when the object has reached a valid state. Use process objects for multi-object consistency. Use events to mark completed conceptual changes.

54.6 When Metadata, Semantic Graph and SQL Change Together

Metadata generation, semantic type materialization, SQL migration and event delivery are separate consistency domains. Publication of an APPLICATION metadata generation is not proof that a `.sem` graph or database shape has switched. For a breaking database facet change, the intended evolution is *EXPAND*, then data *MIGRATE*, then controlled metadata/semantic *SWITCH*, and only later *CONTRACT*. The proposed `PersistenceShape` and migration planner describe the target abstraction, not a verified generic cross-store distributed transaction in the supplied code.

Native semantic type evolution creates a new immutable representation when the structural signature changes; historical instances are not retargeted as a side effect of incremental compilation. Transactional WAL/MVCC behavior in the persistent hypergraph does not provide SQL atomicity by itself. A production design must specify reconciliation, idempotent replay or compensation if commits straddle these boundaries.

54.7 Summary

Identity and consistency are conceptual as well as technical. A logiCells model should make identity explicit, align its facets and choose transaction boundaries that match business intentions.

Chapter 55

Reactive Execution and Event Routing

logiCells applications are not only executed by direct calls. They also react to events. Events make changes observable, allow process models to coordinate participants, allow views to update without owning the business logic, and allow messaging components to propagate work asynchronously.

This chapter explains how event routing contributes to declarative conceptual programming.

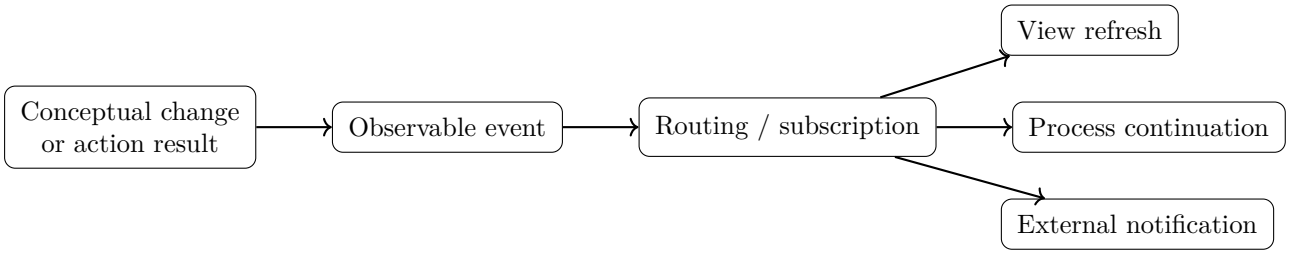


Figure 55.1: Reactive behavior is organized around observable conceptual changes and explicit routing boundaries.

55.1 Events as Declarative Links

An event declaration connects an event class to one or more actions. It can also bind event parameters to action parameters.

```
events:
  - lbs.WorkItem.Started:
      actions:
        - action:
            path: './Action[@name=''OnWorkItemStarted'']'
            params:
              - Source:
                  eventParam: Source
              - Sender:
                  eventParam: Sender
```

Listing 55.1: Event routed to an action

The receiver does not need to be called directly by the emitter. It only declares that it reacts to the event.

55.2 Business Events

A business event states that something meaningful happened in the domain. Examples include process start, process end, message received, role assigned, document validated or work item completed.

Business events should be named from the domain point of view. They should not expose UI implementation details. A good business event name can be understood without knowing which screen will react to it.

55.3 Visual Events

Visual events are used to coordinate views and concrete components. A button action, frame opening, selection change, header update, dataset notification or navigation event belongs first to the visual layer.

Visual events are still declarative. A button can declare that it emits a visual event. A frame can declare that it reacts to a visual event. The logic remains visible instead of being hidden in a hard-coded UI controller.

The page-level dispatcher is the `PageControler`. It receives visual events, notifies the `PageModel` and component `objectBehind` objects, and only escalates the event when the model maps it to an action, process, service, event manager or conceptual container operation. This prevents purely visual interactions from becoming unnecessary server calls or domain mutations.

55.4 Process Events

A process model often emits events at its boundaries. For example, `Start` may emit `lbs.Process.Started`; `Finalize` may emit `lbs.Process.Completed`. Intermediate events can be used to request input, wait for a remote answer or notify a controller.

This pattern makes the process auditable. The process object owns the coordination, but the reactions are declared separately.

55.5 Messaging Events

Messaging events extend the same pattern across asynchronous boundaries. A local action can build a message, send it through the runtime, and let a messaging component dispatch it locally, remotely or both.

A platform message model such as `lgc#custom_message` or `lgc#text_message` can carry state, conversation links, timestamps and sender information. An application may specialize the business workflow, while the asynchronous message dispatch remains a reusable platform mechanism.

55.6 Event Sources and Inheritance

Event declarations may restrict the source of an event. This is useful when a controller must react only to events emitted by a specific process type or object family. Inheritance can also be used when the receiver should react to events emitted by descendants of a conceptual type.

The event class expresses what happened. The source restriction expresses where it happened.

55.7 Dispatching and Propagation

Runtime scripts commonly build and dispatch events explicitly:

```
Event := Self.ObjectsContainer.BuildMessage('lbs.WorkItem.Process.Start');
Self.ObjectsContainer.SendEvent(Event, Self.EventProxy);
```

Listing 55.2: Building and sending an event

Views may create visual events through a page controller:

```
Event := Self.PageControler.CreateVisualEvent(
  'lbs.Application.Navigation.Next',
  Source,
  Sender,
  Self.ObjectsContainer.MetaDataContainer,
  false);
Self.PageControler.DispatchVisualEvent(Event, StopNotification);
```

Listing 55.3: Creating and dispatching a visual event

Both examples follow the same architectural idea: the sender emits a meaningful event, and the runtime routes it to declared receivers.

Chapter 56

Services, Actions and Publication

A logiCells service is not introduced as a separate technical layer that ignores the model. It is usually rooted in an action declared by a concept. The action names a capability, its parameters define an invocation contract, and its publication status indicates that the capability may be exposed beyond the local script that implements it.

This chapter explains the path from an executable action to a service endpoint. It also clarifies a common ambiguity: a published action is not only a visible action in a user interface. It is a model-level capability that can be accessed by other runtime objects, agents, service adapters and web services when the application configuration authorizes such access.

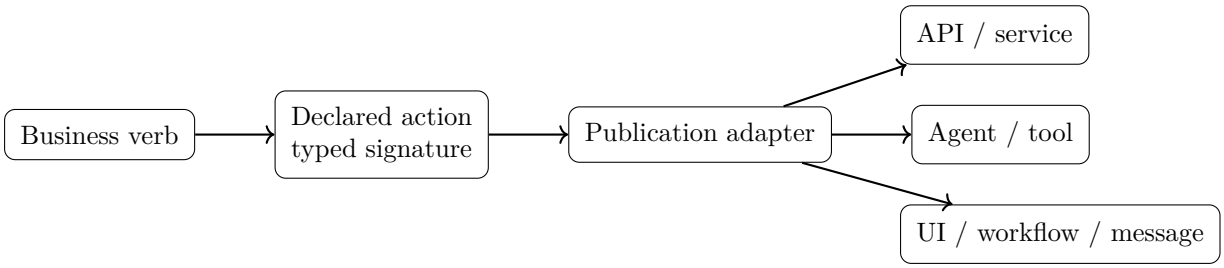


Figure 56.1: One declared capability can be projected through several access mechanisms without duplicating the business operation.

56.1 Action as a Service Boundary

An action is the operational face of a concept. A concept may define data, relations, collections, states, events and views, but an action declares something that the concept can do. When the action is marked as published, the model states that this capability is part of the public contract of the concept.

The following distinction is important.

Internal action An action used by a script, a view model, an event handler or a process implementation. It may remain private to the model.

Published action An action intentionally made available as a callable capability. It can be invoked by other model elements and can be selected for service publication.

Exposed action A published action bound to a concrete access mechanism, such as a web service, service group, agent command or message-based bridge.

This separation allows a model to remain stable even when the external publication strategy changes. The same conceptual action may be used by a local form, a server-side process, a scheduled task, a web service or a remote agent without duplicating the business capability.

56.2 The Declarative Signature

The signature of a published action is declared in the model. It contains the action name, its carrier, its execution mode and its parameters. Parameters are not only implementation details: they define the service contract that external callers must satisfy.

The carrier determines how the service is addressed. An `ObjectItem` action is invoked on a concrete conceptual object. A `ClassItem` action is invoked at concept level and is often used for factory, search, import, publication or administrative capabilities.

56.3 Web Service Access

A published action can be accessed through a web service when a service adapter is configured to expose it. The web service adapter does not invent a separate business API. It maps an external call to the action signature already declared by the model.

The mapping is typically organized around four elements.

1. The target concept, identified by its class id or model path.
2. The target action, identified by its published name.
3. The input parameters, mapped from the request payload.
4. The return parameter or response object, mapped back to the service response.

This approach makes service publication model-driven. The API surface is not a manually maintained mirror of the domain. It is a controlled projection of selected conceptual capabilities.

56.4 Synchronous Service Calls

The simplest publication mode is synchronous. A client calls a web service, the adapter resolves the target concept or object, invokes the published action, waits for the result, and returns the response.

This pattern is appropriate when the caller expects an immediate answer and when the action is short-lived enough to be executed inside the request-response boundary. Typical examples include validation, lookup, creation of a lightweight object, computation of a status or retrieval of a summarized projection.

56.5 Asynchronous Service Bridges

Not every published action should be executed synchronously. Long-running, remote or multi-participant operations are better modeled through a process object or message bridge. In that case the web service may only initiate the process and return an acknowledgement, while the process continues through events and messages.

A common pattern is:

```
External request
  -> published action StartProcess
  -> create process object
  -> emit process-started event
```

```
5   -> return acknowledgement
6   -> process continues through events/messages
```

Listing 56.1: Synchronous entry point, asynchronous continuation

The action remains the service boundary, but the process owns the coordination logic.

56.6 Security and Publication Scope

Publication is not authorization. Marking an action as published states that the capability is callable in principle. Access still depends on service configuration, roles, contextual visibility, authentication, authorization and validation rules.

A robust model therefore treats publication in layers:

1. The action is declared.
2. The action is marked as published.
3. A service adapter exposes the action.
4. Security rules decide who may call it.
5. Validation rules decide whether the call is valid in the current context.

56.7 Design Rules

Published actions should be named after business capabilities rather than transport mechanisms. Prefer `Verify`, `Enroll`, `Start`, `Finalize`, `Import`, `Synchronize` or `Search` to names that describe HTTP or UI concerns.

A published action should have a stable signature, explicit parameter types and a clear return convention. When the action coordinates several objects, introduce a process model rather than hiding orchestration inside a service script.

Chapter 57

Runtime Boundaries and Extension Points

A logiCells application is extensible because the runtime separates model declarations from concrete implementations. This chapter identifies the main boundaries where extension occurs.

57.1 Model Boundary

The model boundary is where concepts, roles, facets, collections, actions, events and views are declared. This boundary should remain stable and readable. It is the public contract of the application model.

57.2 Implementation Boundary

Some model elements are executed by concrete implementation classes. An action may be scriptable or implemented by a registered class. A functional component may be abstract in the model and rendered by a concrete component registered for a platform. A service adapter may expose an action through HTTP or another protocol.

57.3 Publication Boundary

Published actions form a controlled boundary between the runtime and external access. A published action can be called by UI, service, agent or webservice adapter, but the model remains the source of the signature.

57.4 Messaging Boundary

Messaging introduces an asynchronous boundary. A local event can become a message. A remote message can become a local event. The model should declare which events are local, remote or both when that distinction matters.

57.5 Design Rules

Keep business meaning in the model. Put replaceable behavior behind registered implementations. Do not expose internal helper actions as services. Treat webservice publication, messaging and UI activation as adapters over model capabilities.

57.6 Implementation Bindings Are Not Business Concepts

Some current metadata declarations bind a conceptual artifact to a registered runtime implementation, for example a generator or an object implementation selected by a binding identifier. Treat that identifier as an implementation selector. It should not determine the name of the business concept, the ontology verb, the published capability or the user's mental model.

Where stable declarative aliases are available, prefer capability-oriented names and let registration resolve them to concrete implementations. Where the grammar still requires a concrete class name, keep that binding at the runtime boundary and explain it as such. This preserves the direction of abstraction: business concepts select capabilities; capabilities resolve to implementations.

57.7 Summary

Runtime boundaries let logiCells applications evolve. The model defines meaning; implementations, services and components provide concrete execution.

Chapter 58

Runtime Lifetime, Ownership and Shutdown

Lifetime is part of the runtime contract. Failures that look like data, messaging, UI or AI problems often come from an object, callback, worker or native result outliving the resource it depends on.

This chapter keeps the public rules and leaves implementation-language details to the runtime engineering guides.

58.1 Memory Lifetime and Operational Lifetime Are Different

Keeping a reference alive does not mean the resource is operational. A remote channel can still exist after it is closed. A service object can still exist after it is stopped. A resident inference runtime can still exist after its context cache is reset.

Always distinguish:

- **memory lifetime**: whether an object/value may still be referenced;
- **operational lifetime**: whether it is started, connected, subscribed or accepting work.

58.2 Ownership Families

At application level, four families matter:

Managed/reference-counted contracts Holding the public reference keeps the wrapper or contract alive, but does not replace explicit stop/close operations.

Explicitly owned runtime objects The constructor, factory or method contract determines who releases the object.

Client-framework ownership Rich-client components can also participate in UI/component ownership and event subscription lifetimes.

Low-level runtime values Advanced hypergraph/native values may have a runtime-specific retention contract and must not be treated as ordinary host-language objects.

58.3 Service and Worker Shutdown Order

A deterministic shutdown normally proceeds from producers toward shared state:

1. stop accepting new commands and user actions;
2. unsubscribe callbacks and event sinks;
3. stop timers, schedulers and background producers;
4. cancel or drain outstanding work according to application policy;
5. close remote channels and transports;
6. stop services and synchronizers;
7. wait for workers that can still reach application state;
8. release datasets, providers, runtime models and containers;
9. unload modules and unregister global factories last.

Destruction is not a substitute for an explicit stop/close transition when work may still be in flight.

58.4 ABI Ownership

The ABI adds a machine-readable ownership contract. A binding must obey the ownership and release policy in the descriptor rather than guessing from a method name or return type. The detailed ownership vocabulary and release kinds are documented in Chapter 64, “Declarative Models and the Runtime ABI”.

58.5 Resident AI Lifetime

Resident model weights and per-context state have different lifetimes. A new inference context can reset its cache without unloading immutable model weights. Recreating a resident runtime for each token or request defeats the point of residency and can hide lifecycle bugs behind repeated initialization.

The same rule applies to provider-backed local inference: stopping request processing and unloading the model are distinct transitions.

58.6 Rich Client Lifetime

A View, DataSet/DataCursor, asynchronous continuation and page/controller object can have different lifetimes. A client callback must not update a page after the page has been disposed, and a synchronization worker must not retain a client object solely because an operation is still pending. The logiCells Rich Client View Guide gives the client-specific patterns.

58.7 Developer Checklist

Before considering shutdown complete, answer:

- Can any thread or continuation still call application code?
- Can an event source notify an object that is being destroyed?
- Can a remote response arrive after its request context is gone?

- Is a queue, provider or cache being released while a producer still uses it?
- Has every started service received its matching stop/close transition?
- Have owned ABI results been released exactly once?
- Have global registries and native modules been left until their active users are gone?

A deterministic shutdown path is a practical test that ownership has been modeled correctly.

Part VIII

CLI, APIs and External Integration

Chapter 59

Using the cell Command-Line Tool

The command-line examples use `cell`. Check `cell -help` and the version of the Tooling Host installed in your environment before scripting them; command groups and options can change between distributions.

59.1 Architecture

The CLI is a facade over the Tooling Host. Workspace, application, package and metadata business rules remain in tooling services; the CLI must not silently reimplement them. The normal path is JSON-RPC/STDIO to `logicells-tooling`, with logs kept separate from the final result document.

59.2 Global options

```
cell [--json] [--quiet] [--verbose]
     [--tooling <path>] [--tooling-endpoint <url>]
     [--timeout <ms>] [--lgc <file.lgc>]
     [--param name=value] [--workspace-folder <path>] <command>
```

`-json` reserves stdout for a stable machine-readable envelope. Diagnostic logs are routed outside that result stream.

59.3 Start with doctor

```
cell doctor
cell --json doctor
```

The diagnostic reports the actual capability matrix exposed by the selected Tooling Host. A missing capability remains visible rather than falling back to a second implementation in the CLI.

59.4 Workspace, application and package inspection

```
cell workspace inspect [path]
cell application inspect <application>
cell application validate <application>
cell application packages <application>
cell package inspect <package>
cell package validate <package>
```

Application build and compile are distinct commands:

```
cell application build <file.lgc>
cell application compile <file.lgc>
cell application compile plan <file.lgc>
```

The plan surface is public only when the Tooling Host advertises the corresponding capability.

59.5 MODEL authoring and inspection

```
cell declarative inspect <file>
cell declarative validate <file>
cell model file inspect <file>
cell model source inspect <file>
cell model metadata inspect <file>
cell model source convert <file> --format xml|yaml [--output <file>]
```

The CLI also exposes structured field, relation and action add/update/remove commands with `-dry-run`. Prefer these commands for agentic or automated edits because tooling performs the same model resolution and validation as interactive workflows.

59.6 Semantic exploration

The current executable exposes semantic inspection over an assembled application, an H-Logic source, or a project:

```
cell semantic inspect <file.lgc>
cell semantic source inspect <file.hlg>
cell semantic project inspect <path>
cell semantic nodes <application|source|project> <path>
cell semantic node inspect <mode> <path> <node>
cell semantic neighbors <mode> <path> <node>
cell semantic relations <mode> <path> <node>
cell semantic stats <mode> <path>
cell semantic path <mode> <path> <from> <to>
cell semantic export <mode> <path> <node>
cell semantic query <mode> <path> <node>
cell semantic explain <mode> <path> <node>
```

59.7 Scripts, services, runtime and debugger

```
cell script languages
cell script validate <source> [--language <name>]
cell script compile <source> --output <ast> [--language <name>]
cell script inspect <ast>
cell script run <source|ast> [--language <name>]

cell services inspect <service-file>
cell services status <file.lgc>
cell server inspect <file.lgc>
cell server start <file.lgc> [--foreground]
cell server list
cell server status [pid]
cell server stop [pid]
```

Debugger commands operate on runtime process identifiers and include stack, locals, evaluation, continue/step/step-in/step-out, termination and breakpoint set/clear.

59.8 Application publication: MCP and OpenAPI

The current executable exposes symmetric publication compilers for applications and packages:

```
cell mcp compile application <file.lgc> --output <dir> [--generate-doc]
cell mcp compile package <package> --output <dir> [--merge] [--generate-doc]
cell openapi compile application <file.lgc> --output <dir> [--generate-doc]
cell openapi compile package <package> --output <dir> [--merge] [--generate-doc]
```

These commands project the selected application/package contract to protocol-specific artifacts. MCP application artifacts can be hosted by listing the MCP server in the application's `.lgc` Agents configuration. See Chapter 61, "Publishing Applications through OpenAPI and MCP".

59.9 Agent execution and persistent sessions

The one-shot command is:

```
cell agent run <task> [--agent-id <id>] [--subject-id <id>]
               [--purpose <name>] [--profile <name>]
               [--knowledge-source <path>] [--knowledge-mode source|project]
               [--knowledge-answer deterministic|verbalized]
               [--knowledge-query <hlogic>]
               [--knowledge-nested-query <hlogic>]
               [--max-turns <n>] [--max-provider-iterations <n>]
               [--max-iterations <n>] [--interactive]
               [--on-confirmation fail|reject]
```

For persistent execution the CLI exposes:

```
cell agent session open [--agent-id <id>]
cell agent session goal <session-id> <goal> [options]
cell agent session pending <session-id>
cell agent session confirm <session-id> <request-id>
cell agent session reject <session-id> <request-id>
cell agent session snapshot <session-id>
cell agent session evaluate <session-id>
cell agent session close <session-id>
```

Session commands require a persistent `-tooling-endpoint`; the endpoint preserves state across separate CLI invocations. See Chapter 69, "Agent Runtime, Goals and Persistent Sessions" for the lifecycle and knowledge-query options.

59.10 CLI v0.17: Incremental Diagnostics, Transformers and Documentation

Command availability depends on the installed Tooling Host and CLI version. `cell application compile plan` inspects a plan without applying it; `compile` requests an eligible incremental compilation; `build` requests a complete build. MODEL edits driven by source files differ from operations on compiled APPLICATION metadata, which need not rewrite `.model.yaml` files. Check the active command's help before automating it.

The CLI exposes discovery and publication paths additional to the basic scripting and API commands:

```
cell transform list
cell transform capabilities
cell mcp compile application <file.lgc> --output <dir>
cell openapi compile application <file.lgc> --output <dir>
cell doc compile application <file.lgc> --output <dir> --format xml|html
```

The XML documentation pipeline is exposed through Tooling. Check `--format xml|html` and the generated documentation on the version deployed. Publishing OpenAPI or MCP descriptors

does not automatically apply the Agent confirmation workflow to calls coming through an external endpoint; validate its authorization separately.

The supported runtime-debug facade connects to `sdlgc` with `--ast-debug`. Use the logical `processpid` returned by the debug runtime, not the host OS process identifier. Canonical new command forms include `cell debug break add`, `break clear`, `step-in` and `step-out`. Interactive debug of an embedded YAML script requires the additional source mapping described in Chapter 75.

59.11 Do Not Conflate Agent and Developer-Agent Namespaces

This guide describes `cell agent run` and `cell agent session`. Workspace-development commands and additional agent profiles, where provided, must be checked against the installed CLI and Tooling registry before creating automation that relies on them.

59.12 Contract rule

Treat the executable help and Tooling Host capability negotiation as the operational authority for command availability. Documentation may describe a planned command only when it is clearly marked as planned; it must not present an unadvertised tooling capability as implemented.

Chapter 60

API and Service Publication

For the metadata-to-runtime mechanics and advanced service families, see Chapter 48, “Declarative Runtime Services and Engine Patterns”.

An API in logiCells should be understood as a published projection of conceptual capabilities. The service surface is not a parallel model. It is a controlled access layer over concepts, objects, actions, events and processes.

60.1 Purpose

This chapter describes how APIs and services should be designed from published model capabilities. It focuses on the relation between action signatures, service endpoints, request and response mapping, and process-oriented service design. The concrete OpenAPI and MCP compilation and hosting workflow is documented in Chapter 61, “Publishing Applications through OpenAPI and MCP”.

60.2 From Action Signature to API Operation

A published action provides a natural operation descriptor. Its name becomes the operation name, its carrier becomes the target resource or service context, its parameters become request fields, and its return parameter becomes the response.

```
ClassItem: lbs#account_service
Action:    SearchAccount
Params:    Query:String, MaxCount:Integer, Result:List

Possible endpoint:
POST /services/lbs/account-service/SearchAccount
```

Listing 60.1: Conceptual action to web service operation

The exact endpoint format is an adapter decision. The conceptual operation remains the same.

60.3 Request and Response Mapping

Request fields must be mapped to declared parameters. Primitive types are mapped directly. Conceptual types are resolved through ids, class ids, object references or serialized projections. Return values may be primitive values, object references, lists, status objects or message acknowledgements.

60.4 Service Groups

Service groups organize endpoints and integration responsibilities. A service group may expose synchronous web services, messaging bridges, remote agents or specialized adapters. The group should not redefine business semantics. It should route external calls to conceptual actions or messages.

60.5 Published Actions over Web Services

A web service endpoint can expose a published action directly. This is the most transparent model because the external API follows the conceptual capability. The adapter validates the request, checks access, resolves the target, invokes the action and returns the result.

60.6 HTTP Virtual Pages and Route Selection

HTTP publication can also define virtual pages. A `serviceGroupDefinition` with `type: HTTP` carries `virtualPages` rules. In the documented example, `GET /users` matches a `method/path` entry, selects the registered `PathRequestWebPage` handler by its ABI name, and applies the declared conceptual `rewrite` target. HTTP page-extension resources use the `.lgp` suffix. The route declaration and the page-resource filename are separate concerns. See Chapter 48, "Declarative Runtime Services and Engine Patterns" for the YAML form and request-selection chain.

60.7 Process-Oriented APIs

Long-running operations should not be modeled as single blocking API calls. The API should expose a published action such as `Start`, `Submit`, `Request` or `Finalize`. That action creates or updates a process object and returns an acknowledgement or process reference.

60.8 Machine-Readable Contracts: OpenAPI and MCP

OpenAPI and MCP are publication surfaces over the same conceptual runtime. They should not invent a second type system. A generated OpenAPI schema and an MCP tool `inputSchema` should preserve the distinctions already present in the model: primitive values, typed object references, relation roles, required parameters and supported formats.

This gives a practical conformance rule:

```
concept/action metadata
-> runtime parameter and relation contract
-> OpenAPI request/response schema
-> MCP tool inputSchema/output contract
-> adapter validation
-> runtime execution
```

Listing 60.2: One contract projected to several publication surfaces

If a role path is required to write or resolve a relation, the machine-readable contract should expose enough structure for a client to discover it. Requiring a developer to infer hidden role paths from existing projects means the generated contract is incomplete.

MCP deserves the same discipline as HTTP. A tool call must be validated against the declared tool schema before dispatch. Unknown arguments, missing required arguments, malformed date-time values, unresolved class identifiers and invalid object-reference shapes should

be rejected before the underlying action is executed. An MCP manifest is therefore not only a menu of tools; it is an executable boundary contract.

For `ClassId`, publication adapters should prefer exact resolution. If a value is syntactically a class identifier, silently falling back to a class name can hide case or namespace errors and create different behavior between OpenAPI, MCP and native calls.

60.9 Error and Validation Semantics

Errors should be expressed at the boundary where the invalid value is first known. Technical errors belong to the service adapter. Validation errors belong to the action or process. Business exceptions should be represented by states, events or structured response objects when the caller must handle them.

A publication adapter should return diagnostics precise enough to identify the failed contract: operation/tool, parameter or role, received value, expected type or format, and the unresolved `ClassId` when relevant. Do not convert an invalid typed reference into an empty relation and do not accept an invalid `DateTime` only to fail later before persistence.

60.10 Design Rules

Design the API from the model outward. Publish stable actions. Keep service adapters thin. Use processes for long-running operations. Use messages for asynchronous continuation. Use explicit types and return conventions. Avoid exposing internal table names or view-specific details.

Chapter 61

Publishing Applications through OpenAPI and MCP

logiCells can project an application or package into machine-readable publication artifacts for both OpenAPI and MCP. These are two protocol projections of the same application model; they are not separate business models.

The developer workflow is deliberately explicit:

```
application / package
        |
        +--> cell openapi compile ... --> OpenAPI publication artifacts
        |
        +--> cell mcp compile ...     --> MCP publication artifacts
```

The projection starts from the application and its published capabilities. Protocol-specific artifacts are generated into an output directory and can then be hosted by the corresponding runtime service configuration.

61.1 Publication Starts with the Application Contract

An external protocol should expose capabilities that are already meaningful in the conceptual model. Published actions are the primary operational boundary. Their declared names, parameters, types and publication status form the contract from which external protocol surfaces are generated.

For example, action publication can be controlled from the CLI:

```
cell application action add <file.lgc> <classId> <name> --published
cell application action update <file.lgc> <classId> <name> --published
cell application action update <file.lgc> <classId> <name> --unpublished
```

The same distinction is available on MODEL files through `cell model action add/update`. Publication therefore remains a model decision before it becomes a protocol decision.

61.2 Compiling an Application for MCP

The current CLI exposes MCP compilation for complete applications and packages:

```
cell mcp compile application <file.lgc> --output <dir> [--generate-doc]
cell mcp compile package <package> --output <dir> [--merge] [--generate-doc]
```

A concrete application build can be produced with a project-local path:

```
cell mcp compile application /path/to/app.lgc \
  --output /path/to/build/mcp
```

The command writes the MCP projection to the selected output directory. Use `-generate-doc` when the generated documentation is required together with the publication artifacts. For packages, `-merge` controls the package-oriented merge behavior exposed by the current executable.

61.2.1 Hosting the generated MCP surface

The generated MCP surface is made available by declaring the MCP server in the application's `.lgc` configuration used by Agents. When the configured application is started, the Agents runtime starts the listed MCP server. This gives an external MCP client a protocol boundary into the selected, published application capabilities.

The lifecycle is therefore:

```
.lgc application
   -> cell mcp compile application
   -> generated MCP artifacts
   -> MCP server listed in the .lgc Agents configuration
   -> Agents runtime starts the server
   -> external MCP clients connect
```

The exact `.lgc` server stanza is versioned configuration data. Use a canonical example or schema that matches the installed release; do not copy arbitrary internal fixtures or older server declarations. Validate the application's configuration before exposing an endpoint.

61.3 Compiling an Application for OpenAPI

OpenAPI follows the corresponding application/package publication workflow:

```
cell openapi compile application <file.lgc> --output <dir> [--generate-doc]
cell openapi compile package <package> --output <dir> [--merge] [--generate-doc]
```

For example:

```
cell openapi compile application /path/to/app.lgc \
  --output /path/to/build/openapi
```

OpenAPI and MCP should be treated as sibling projections. A developer should not encode business rules once for OpenAPI and again for MCP. The model and published actions remain authoritative; protocol compilers derive their external contracts from that source.

61.4 Application and Package Publication

Both protocol families support application and package inputs. Choose the input according to the boundary being published:

Application compilation Use when the external contract belongs to the assembled application described by a concrete `.lgc` configuration.

Package compilation Use when the publication contract is built from a reusable package boundary. Package compilation also exposes the optional `-merge` behavior.

Before publishing an application, the following commands are useful verification steps:

```
cell application validate <file.lgc>
cell application inspect <file.lgc>
cell application packages <file.lgc>
cell application models <file.lgc>
```

For a package:

```
cell package validate <package>
cell package inspect <package>
cell package item inspect <package> <name>
```

61.5 The Two MCP Directions

MCP support must be read in two distinct directions.

logiCells as an MCP server An application or package is compiled to an MCP publication surface and hosted through the runtime configuration. External MCP clients and agent orchestrators can then access the published application boundary.

logiCells as an MCP client The Agent runtime can discover and invoke capabilities from external MCP servers. This path is governed independently and can distinguish read-only from effectful operations.

These two directions solve different integration problems. Server publication exposes logiCells capabilities outward; client operationalization allows logiCells agents to consume external tools inward.

61.6 Type and Validation Discipline

OpenAPI request/response schemas and MCP tool schemas are protocol representations of an existing typed contract. Publication adapters should preserve, rather than weaken, the type information available in the model.

In particular:

- required parameters should remain required at the protocol boundary;
- malformed typed values should fail before the underlying action is dispatched;
- conceptual identifiers should retain exact identity semantics;
- a published action should not become callable under a looser protocol contract than its native runtime contract;
- generated documentation should be regenerated when the published application contract changes.

61.7 Recommended Developer Workflow

A repeatable publication workflow is:

1. Validate the application or package.
2. Inspect the actions intended for publication and verify their `published` status.
3. Compile the OpenAPI and/or MCP projection to a clean output directory.

4. Generate protocol documentation when useful with `-generate-doc`.
5. Configure the runtime service boundary in the application's `.lgc` file.
6. Start the application/runtime and verify the published surface with an external protocol client.
7. Recompile after changes to published action signatures or relevant model metadata.

61.8 Protocol Projection Is Not Authorization

Generation of a protocol surface does not by itself grant permission to every caller. Publication, runtime security, policy and effectful-operation controls remain separate concerns. A capability can be present in a generated contract and still require runtime authorization or confirmation before execution.

This distinction becomes especially important with MCP because the caller may itself be an agent. Treat discoverability, authorization and execution as separate stages.

61.9 Inbound MCP Calls and Runtime Authority

For inbound MCP, keep three questions separate:

1. Is an action published and therefore visible in the generated MCP surface?
2. Is the caller allowed to request that operation in the current execution context?
3. If the operation is effectful, is confirmation or other runtime evidence required before dispatch?

Compiling and hosting an MCP endpoint are separate from authorizing an effectful inbound action. Test permissions and any required confirmation directly through the published endpoint. Review audit and error behavior before granting external access.

Therefore this guide does not claim an inbound MCP call is governed merely because the action is published. The normative contract for that path is specified in Chapter 70, "Governed Agent Execution Contract". A release should mark inbound policy enforcement, confirmation gating, one-shot dispatch and audit as verified only when tests establish those behaviors for the server path.

61.10 Reference Configuration Requirement

An Available server surface should have a schema-valid, copyable configuration example. The canonical example must come from a version-controlled starter or configuration schema accepted by the current runtime. It should show, at minimum, the MCP publication artifact, endpoint/transport settings where applicable, and the Agents entry that causes the server to start.

This revision deliberately does not manufacture that stanza from prose. Once the canonical starter is present in the distribution, the example should replace this paragraph and be exercised by the documentation build or release test.

Chapter 62

Declarative External API Adapters

Many logiCells applications have to consume external APIs. The important point is that the external API should not become the model. It should be wrapped as a declarative adapter that exposes requests, responses, pagination, errors and services as conceptual objects.

This pattern allows an application to connect to external systems while keeping a stable conceptual surface for providers, datasets, actions and processes.

62.1 External APIs as Conceptual Boundaries

An external API adapter should be modeled as a boundary between the logiCells runtime and a remote system.

```
logiCells action or provider
  -> API service model
  -> request concept
  -> remote endpoint
  -> response concept
  -> collection provider or process update
```

Listing 62.1: External API adapter chain

The adapter should describe the contract. It should not spread remote URLs, query parameters, headers or response pagination rules through visual components or business processes.

62.2 Request, Response and Service Concepts

A robust adapter usually separates three concepts.

Concept	Responsibility
Request	Declares URI, method, headers, parameters and payload.
Response	Declares status, success flag, error information, data payload and pagination metadata.
Service	Connects a request type to a response type and provides the executable integration boundary.

This separation makes the integration inspectable. It also makes it possible to test requests and responses independently from the visual layer.

62.3 A Generic Paginated API Pattern

The following example uses a generic business resource. It shows only the public pattern and not a client-specific API.

```
models:
  concepts:
    - RemoteWorkItemRequest:
        type: Entity
        id: lbs#remote_work_item_request
        facets:
          - uri:
              type: hypergraph
              fields:
                - Endpoint:
                    type: String
                    kind: attribute
                - Page:
                    type: Integer
                    kind: attribute
                - ItemsPerPage:
                    type: Integer
                    kind: attribute
                - Filter:
                    type: String
                    kind: attribute
    - RemoteWorkItemResponse:
        type: Entity
        id: lbs#remote_work_item_response
        facets:
          - main:
              type: hypergraph
              fields:
                - Success:
                    type: Boolean
                    kind: attribute
                - CurrentPage:
                    type: Integer
                    kind: attribute
                - TotalPages:
                    type: Integer
                    kind: attribute
                - TotalItems:
                    type: Integer
                    kind: attribute
                - Items:
                    type: lgc#list
                    kind: slot
    - RemoteWorkItemService:
        type: Service
        id: lbs#remote_work_item_service
        facets:
          - main:
              type: hypergraph
              fields:
                - Request:
                    type: lbs#remote_work_item_request
                    kind: slot
                - Response:
                    type: lbs#remote_work_item_response
                    kind: slot
```

Listing 62.2: Paginated API adapter in YAML model syntax

Here, `kind` is used only on model fields declared inside facets. It classifies conceptual roles. It is not used on datasets, datasources or cursors.

62.4 From API Adapter to Collection Provider

The adapter becomes especially useful when it is connected to a collection provider. The provider hides paging, retries, authentication, refresh and synchronization details from the view.

```
UXfc table or lookup component
  -> DataSource
```

```
3    -> DataSet
4    -> RemoteWorkItemProvider
5    -> RemoteWorkItemService
6    -> RemoteWorkItemRequest / RemoteWorkItemResponse
7    -> remote API
```

Listing 62.3: Remote API behind a collection provider

The visual component remains stable. The provider decides whether data is loaded from a local cache, refreshed from the remote API, paginated lazily, or synchronized in the background.

62.5 Errors, Status and Traceability

A response concept should usually expose more than the returned data. It should also expose success status, technical status, business error messages, timestamps, trace identifiers and pagination state when available.

This information can be used by diagnostics, retry policies, dashboards, human validation workflows and AI assistants.

62.6 Design Rules

Model external APIs as explicit adapters. Separate request, response and service concepts. Keep pagination metadata visible. Connect remote APIs through providers instead of visual components. Escalate failures as events, messages or process states when human or asynchronous handling is required.

Chapter 63

Integration with External Systems

logiCells applications rarely live in isolation. They access databases, exchange messages, call services, receive external events, synchronize local and remote state, and may delegate execution to agents. Integration is therefore part of the platform design, but it should not break the conceptual structure of the model.

The central rule is simple: external systems should connect to conceptual capabilities, not to accidental implementation details.

63.1 Integration as a Boundary Problem

A logiCells model contains concepts, objects, relations, actions, events, views and processes. An external system should be attached at one of these explicit boundaries. Directly coupling external code to internal tables or visual components creates brittle integration.

The usual integration boundaries are:

Database facets Used to persist and project conceptual fields into relational or structured storage.

Published actions Used to expose business capabilities through services or adapters.

Events Used to notify state changes and coordinate loosely coupled participants.

Messages Used to cross process, device or network boundaries asynchronously.

Agents Used to execute tasks, observe changes or coordinate remote work.

63.2 Synchronous Integration

Synchronous integration is appropriate when the caller needs an immediate result. The most common form is an external call to a published action exposed by a service adapter. The action executes in the conceptual runtime and returns a typed result.

This pattern preserves the model as the source of truth. The external system sees a service contract, but the runtime executes a conceptual capability.

63.3 Asynchronous Integration

Asynchronous integration is appropriate when latency, external dependency, human interaction or multi-step coordination makes immediate completion undesirable. In that case, the external call should create or update a process object, send a message, or emit an event.

The asynchronous pattern is especially useful for:

- remote messaging;
- offline synchronization;
- long-running workflows;
- interactions with agents;
- human validation;
- event-driven UI updates.

The platform messaging layer belongs to logiCells/logiCells. Domain workflows may specialize how messages are used, but the mechanism of message declaration, dispatch, conversation and asynchronous coordination is reusable.

63.4 Database Integration

A database facet is a projection of conceptual fields. It should not be confused with the concept itself. The same conceptual field may appear in the hypergraph facet as a conceptual role and in the database facet as one or several persisted columns. Linked objects are typically flattened through role paths such as `Owner.Id`, `Owner.ClassId` or `Owner.FormalName`.

This makes persistence explicit while keeping the conceptual model above the database schema.

63.5 Message and Event Integration

Events express observable changes inside the runtime. Messages allow those changes to cross technical boundaries. A message may be persisted, routed, delivered later, associated with a conversation, or attached to a workflow state.

A good integration design keeps these levels separate:

```
Conceptual event  = something happened in the model
Message           = transportable representation of a change or request
Conversation      = grouping of related messages and participants
Process           = coordination object that reacts to events/messages
Service adapter   = external entry point into the model
```

Listing 63.1: Integration levels

63.6 Agents and External Execution

Agents may observe the model, execute actions, consume messages, publish events or call external services. They should not bypass conceptual semantics. An agent that modifies the system should do so by invoking actions, creating objects, posting data or emitting events through the runtime.

63.7 Design Rules

Prefer published actions for synchronous integration. Prefer events and messages for asynchronous integration. Prefer process objects when several business objects participate. Prefer database facets for persistence, not as the main semantic API. Prefer agents when integration requires observation, automation or delayed execution.

Chapter 64

Declarative Models and the Runtime ABI

The runtime ABI is the typed integration boundary between logiCells and external host code. It is deliberately below the conceptual model: concepts, relations, actions and permissions define the business contract; the ABI defines how external code invokes runtime capabilities safely and how values and object lifetimes cross that boundary.

64.1 Current Availability

The current logiCells distribution provides ABI bindings for **Python** and **C#**. These bindings expose the same ABI contract while mapping lifetime and value rules into the conventions of each language.

Other language bindings must not be assumed to be available unless they are explicitly documented as supported by the distribution in use. In particular, this guide does not claim Java/JNI/JNA support.

64.2 Stable Invocation Identity

An invocation identifier contains a module id, type id, method id and call kind. The call kind distinguishes constructor, static, instance and other invocation forms represented by the ABI. ABI manifests in the current source use schema version 2, ABI version 1 and an explicit identifier contract.

64.3 Value Model

The ABI value model covers null, Boolean, integer and floating-point values, strings, handles, arrays, records, pointers, callbacks, fixed-width numeric values and byte buffers. A value descriptor also carries the information required to marshal the value correctly, including type identity, size, nullability, ownership and release policy.

A method descriptor combines:

- invocation identity;
- qualified and canonical signature;
- receiver requirement;
- argument record size and alignment;

- ordered parameter descriptors;
- result descriptor.

64.4 Ownership Is Part of the ABI Contract

Ownership is explicit metadata. It is not inferred from a method name, return type or naming convention.

For ordinary Python and C# binding users, the two cases encountered most often are:

`borrowed` The returned object remains owned by the runtime or by another runtime object. The caller may use the reference according to the documented lifetime but must not release the native object as if it owned it.

`owned_by_caller` Ownership is transferred to the caller. The language binding is responsible for releasing the underlying native object when the wrapper is closed or disposed.

This distinction prevents both sides of the same failure class: a borrowed object must not be released by the caller, while an owned result must not be silently treated as borrowed and leaked.

> Ownership is explicit ABI metadata. Do not derive it from names such as `Build`, `Create`, `Clone` or `Snapshot`.

64.5 Complete Ownership Vocabulary

Binding authors and runtime-extension authors need the complete ABI vocabulary, not only the two common wrapper-facing cases. The current ABI contract defines six ownership values:

ABI ownership	Meaning at the boundary
`value`	The ABI value is carried by value; no object-lifetime transfer is implied.
`caller`	Storage or lifetime responsibility is associated with caller-provided state according to the descriptor and invocation layout.
`owned_by_caller`	The result/resource is transferred to the caller and must be released by the caller according to the release policy.
`owned_by_module`	The module retains ownership; the caller must not destroy the underlying resource as caller-owned state.
`borrowed`	The caller receives a non-owning reference whose validity is bounded by the owner documented by the API.
`retained`	The boundary records a retained lifetime relationship; binding/runtime code must preserve the matching retain/release contract rather than treating it as a plain borrowed value.

The public enum names in source are implementation identifiers; manifests and generated bindings expose the corresponding stable ownership semantics. A binding generator must preserve these distinctions even if a host language presents them through a smaller wrapper API.

64.6 Release Policies

The current ABI contract defines four release kinds:

`none` no release operation is associated with the value;

moduleRelease release through the module's release function;

moduleFree release/free through the module's free function for the value shape governed by that contract;

method release through an explicitly described method-level contract.

Ownership and release kind are separate fields. A binding must read both. Do not assume that every caller-owned result uses the same release operation.

64.7 Python Binding

The Python binding preserves ABI ownership in its wrappers. An owned native result is wrapped as an owning reference; a borrowed result is wrapped as a non-owning reference.

For an owning wrapper, calling `close()` releases the native object through the descriptor-selected release path. Repeated calls to `close()` are required to remain safe and must not cause a double release.

The recent ownership validation specifically checks that caller-owned results are emitted as owning wrappers, that borrowed results remain non-owning, and that closing an owning wrapper reaches the module release operation exactly once.

64.8 C# Binding

The C# binding follows the same contract through handle ownership. Wrappers for results transferred to the caller own their native handle; wrappers for borrowed results do not.

For an owning wrapper, `Dispose()` releases the underlying native object. Repeated disposal is required to remain safe and must not release the same native object twice.

The generated C# wrappers distinguish owning references from borrowed returns. Consult the generated contract for the selected release and ensure that returned class-like values are released only by the party that owns them.

64.9 Parameters and Write-Back

Each parameter descriptor records its name, source type, ABI value descriptor, byte offset and direction (`in`, `out` or `in/out`). The execution contract supports write-back of updated `out` and `in/out` values.

The ABI supports documented scalar, string, array and interface-handle shapes. Validate write-back, handle reassignment and object-result lifetime when integrating the generated binding with application code.

64.10 Native and RPC Placement

The ABI type system distinguishes native and RPC access modes. Placement is an execution concern: moving a capability across a process boundary must not require a second definition of its conceptual business meaning.

Do not infer host-platform coverage from this abstraction. Runtime availability and ABI-loader certification are separate questions.

64.11 ABI Manifests and Target Information

Generated `*.abi.json` manifests describe modules, types, methods, parameters, results and target characteristics. Target information includes platform/architecture properties such as pointer size, endianness and record packing.

These characteristics matter operationally. Two ABI artifacts with the same logical method names are not necessarily binary-compatible if pointer sizes, record layouts or packing conventions differ.

64.12 Platform Scope of Native ABI Loading

The runtime is available on Windows, Linux and macOS. Native-module loader support remains a separate platform-specific concern; do not infer availability of a particular loader from general runtime availability.

Python and C# are the currently available bindings. For a deployment that requires a specific binding on Linux or macOS, verify the loader/runtime package supplied for that platform rather than assuming parity from the runtime-wide platform statement.

This distinction is deliberate and should remain visible until platform-specific ABI loading is covered by the same executable validation gates.

64.13 Validated ABI Behavior

The Python and C# bindings follow the published ABI value and ownership contracts. When implementing an integration, pay particular attention to:

- module initialization and shutdown;
- object handles, type identity and ownership;
- scalar values and UTF-8 strings;
- primitive, nested and interface arrays;
- managed, unmanaged and nested records;
- `out` and `in/out` scalar, string and array values;
- interface handles and interface reassignment;
- borrowed and caller-owned object results;
- release behavior and idempotent close/dispose for owned results.

This list describes validated ABI behavior, not a promise that every possible host-language construct is automatically supported. When adding a new ABI surface, verify the manifest contract and run the corresponding binding/runtime tests.

64.14 Developer Rule

Model the business contract first, expose a stable capability second, and use ABI descriptors only at the operational boundary. Python and C# wrappers are integration projections of the runtime contract; they must not become the vocabulary of the conceptual model.

Chapter 65

Publication Design Rules

Publication is the act of making a model capability available outside its immediate internal context. A capability may be published to a user interface, a webservice, an agent, a process, or a messaging channel.

65.1 Publish Capabilities, Not Internals

A published action should correspond to a meaningful capability: validate a credential, create a message, start a process, finalize a work item, retrieve an authorized projection. Avoid publishing low-level helper actions that expose implementation details.

65.2 Use Typed Signatures

The parameter list is the service contract. Use clear names, explicit types and a single return role when possible. Conceptual types are preferable when the parameter represents a concept.

65.3 Separate Publication Channels

The same action may be accessible through a view, an agent or a webservice, but the surrounding publication policy may differ. A webservice needs authentication, request mapping and error mapping. A view needs visibility and interaction rules. An agent needs an execution envelope.

65.4 Versioning

Published actions are contracts. Renaming parameters, changing return types or altering side effects should be treated as a versioned change. Prefer adding a new action or a new publication profile when the contract must evolve.

65.5 Summary

Publication should expose stable conceptual capabilities. The model remains the source of the contract; channels are adapters.

Part IX

logiCells and AI

Chapter 66

Conceptual Memory for AI Applications

AI applications need memory, but not all memory is equally useful. A text archive can be searched, but it does not directly expose objects, roles, constraints, relations, actions or process states. logiCells provides a conceptual memory in which information is structured as executable concepts.

66.1 Purpose

This chapter explains why conceptual memory is a strong foundation for AI applications. It also shows how language models, agents and automation components can use logiCells models without replacing the runtime semantics.

66.2 Beyond Document Retrieval

A retrieval system can find relevant text. A conceptual memory can additionally tell which object is involved, which relation connects it to other objects, which actions are allowed, which state the process is in, and which events have occurred.

For AI, this difference matters. The model provides grounding. The AI can reason over structured concepts and propose operations, while execution remains controlled by published actions and runtime validation.

66.3 Structured Context for AI

A conceptual memory can provide an AI with structured context:

- concept definitions;
- object state;
- relations and role paths;
- process states;
- event traces;
- available published actions;
- view-specific projections;
- security and visibility context.

66.4 Actions as Safe Execution Handles

An AI assistant should not directly mutate storage. It should request or invoke capabilities exposed by the model. Published actions become safe execution handles. They define what can be done, with which parameters, and under which validation rules.

66.5 Conceptual Memory and Explanations

Because the model contains explicit concepts, actions and events, AI-generated explanations can be grounded in model elements. Instead of saying that “the system decided”, an assistant can refer to the process state, the rule, the event or the action that produced the result.

66.6 Design Rules

Expose structured context to AI, not unrestricted internal data. Let AI propose actions, but keep execution mediated by the runtime. Use process states for long-running decisions. Use logs and events to make AI-assisted actions auditable.

Chapter 67

Positioning logiCells in the Neuro-Symbolic Design Space

The expression *neuro-symbolic* covers several architectures that combine learned numerical representations with explicit symbolic structures. These architectures do not all solve the same problem, and they do not assign the same authority to their neural and symbolic components. A platform should therefore state precisely where it is positioned instead of using the term as a broad synonym for "AI plus rules."

logiCells is designed around an executable conceptual hypergraph. Types, roles, polyadic relations, rules, observations, events, provenance and runtime traces coexist in that hypergraph. Neural computation is introduced as a second mode of representation and search over the same conceptual memory. This gives logiCells two complementary neuro-symbolic positions:

1. **governed compositional neuro-symbolic integration** at the application boundary, where symbolic structures produce neural inputs and govern the materialization of neural outputs;
2. **Symbolic[Neuro] reasoning** inside the deduction engine, where a formal reasoner calls a neural intuition service to prioritize its search without delegating validity.

The notation `Symbolic[Neuro]` follows the architectural vocabulary popularized by Henry Kautz in his neuro-symbolic taxonomy. The taxonomy is used here as a design map, not as a maturity ladder. A tighter mathematical coupling is not automatically superior to a composition that preserves auditability, replaceability and formal guarantees.

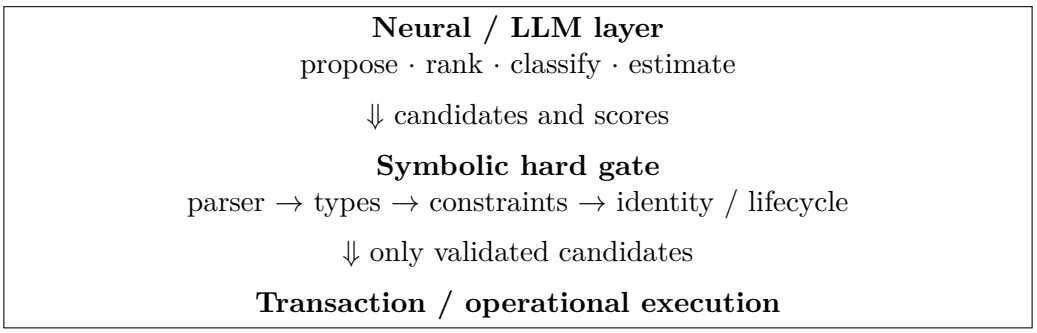


Figure 67.1: Governed neuro-symbolic composition: neural components influence preference, while symbolic validation remains authoritative for validity.

67.1 The Kautz Taxonomy as an Architectural Map

Kautz's taxonomy distinguishes recurring ways in which symbolic and neural systems can be combined. This chapter follows the architectural vocabulary introduced in the AAAI Robert S. Engelmore Memorial Lecture and its published version, while using the compositional vocabulary of van Harmelen and ten Teije where it gives a more precise description of modular boundaries. The central architectural questions are:

- Which component receives the original input?
- Which component performs reasoning or search?
- Which component produces the externally meaningful result?
- Does symbolic knowledge generate supervision for learning?
- Does the neural component merely guide a symbolic procedure, or does it participate in the semantics of the conclusion?
- Are symbolic structures compiled into a neural architecture, or do both components remain operationally distinct?

The taxonomy is useful because it prevents a false binary choice between a purely symbolic engine and a fully differentiable one. Several valid forms of integration exist between those extremes.

67.1.1 Symbolic–Neural–Symbolic Pipelines

A symbolic input can be encoded numerically, processed by a neural model and decoded back into a symbolic output. Many language and embedding pipelines have this general shape. The symbols at the boundaries do not imply that the internal computation performs explicit symbolic reasoning.

logiCells supports this pattern when a conceptual projection is transformed into dense tensors and a model returns labels, scores or embeddings. However, this pattern alone is not the main architectural claim of the platform.

67.1.2 Neural Perception Followed by Symbolic Reasoning

A neural component may convert unstructured or perceptual data into symbols that a formal system can reason about. A document classifier, entity extractor or vision model can propose typed observations and relations. H-Logic and hypergraph transforms can then validate, combine and propagate those symbolic structures.

logiCells can use this pattern for external models, local language models and structured extraction. The important boundary is that neural output enters the hypergraph as a qualified observation or proposal, not as an unquestioned fact.

67.1.3 Symbolic Systems Generating Neural Supervision

A symbolic engine can generate examples, labels, constraints or synthetic cases used to train a neural model. logiCells's conceptual models and H-Logic rules can eventually support this pattern by producing training datasets and deduction traces.

This direction is especially relevant for deduction guidance. Successful and unsuccessful proof traces can become self-supervised examples for learning which premises, rules or branches are useful.

67.1.4 Symbolic[Neuro]

In the `Symbolic[Neuro]` pattern, the symbolic procedure remains the outer algorithm and calls a neural component as a subroutine. AlphaGo is a familiar illustration: the rules of the game and tree search remain explicit, while learned evaluations guide which positions deserve attention.

This is the principal target for neural intuition in logiCells:

```
formal goal and complete candidate set
    -> neural scoring or ranking
    -> reordered symbolic search
    -> H-Logic proof or formal failure
```

Listing 67.1: Symbolic reasoner with neural intuition

The neural model influences *where the reasoner looks first*. It does not determine *what the reasoner is allowed to conclude*.

67.2 logiCells's First Position: Governed Composition

The first implemented neuro-symbolic boundary is compositional and governed. The current framework also contains an executable H-Logic candidate lifecycle: a candidate carries origin and request provenance, is validated independently of graph activation, and can reach operational knowledge only through the controlled transaction/commit path. Parsing a proposal is therefore not equivalent to activating knowledge.

```
conceptual hypergraph
    -> conceptual or dense tensor projection
    -> neural or tensor computation
    -> typed observation or proposal
    -> H-Logic, transforms and governance
    -> accepted, rejected or pending materialization
```

Listing 67.2: Governed compositional cycle

The implementation-level tensor-program material is maintained as an advanced source chapter in this repository rather than in the business-developer reading path. The architectural contract needed here is summarized by the following properties:

Symbolic source of identity Tensor rows, axes and topology positions remain traceable to concepts and roles.

Explicit numerical derivation The tensor is a projection of a selected conceptual state, not a replacement for that state.

Qualified output A score or prediction is represented as an observation, hypothesis or proposed relation.

Governed materialization Rules, transforms, policies or human validation determine whether the proposal affects operational knowledge.

Replaceable execution The neural runtime may evolve without changing the conceptual contract at the boundaries.

This form of integration is deliberately loose at the level of optimization and deliberately strong at the level of semantics and governance. It is particularly suitable for long-lived business systems where traceability, model replacement and human supervision are mandatory.

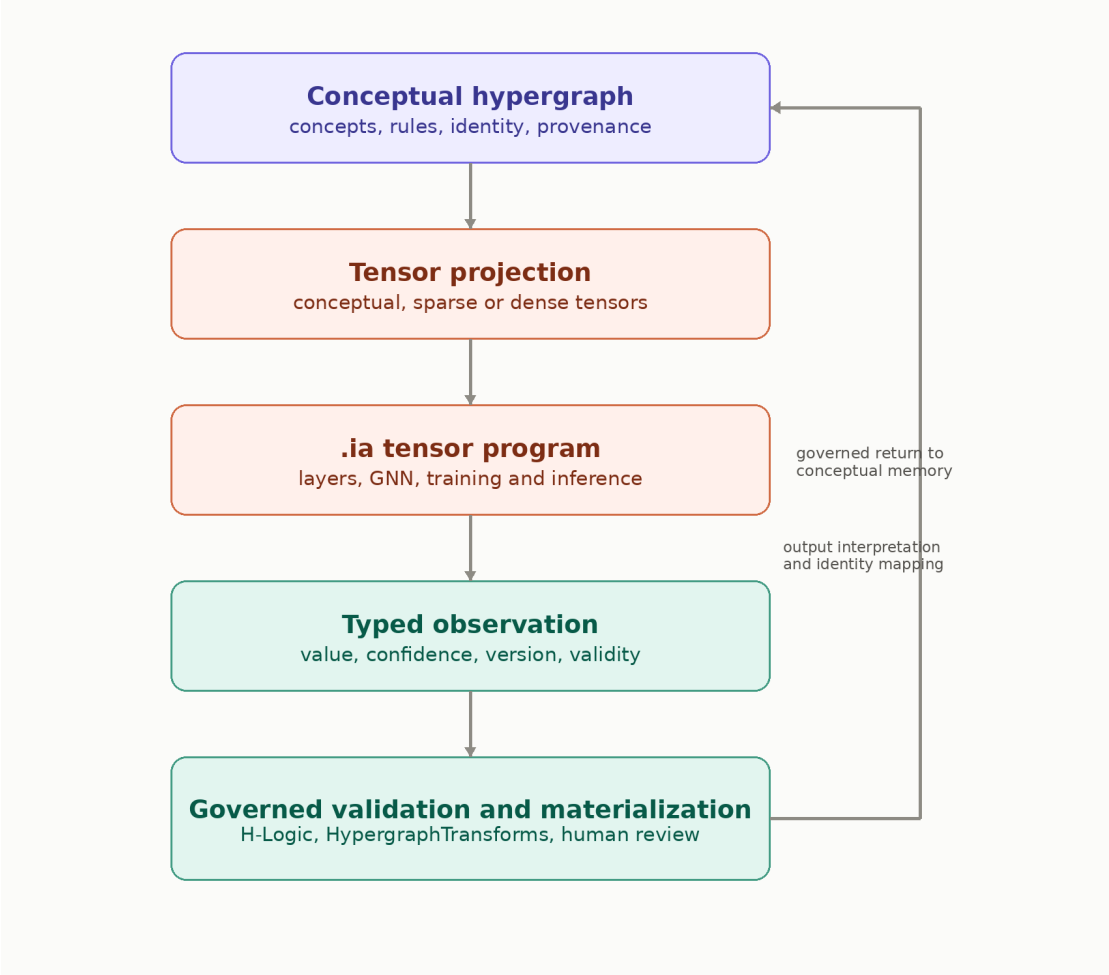


Figure 67.2: Governed compositional neuro-symbolic integration. The tensor program produces a typed observation; H-Logic, hypergraph transforms and human review govern whether and how it returns to conceptual memory.

67.3 logiCells's Second Position: Symbolic[Neuro] Deduction

The second position moves neural computation from the periphery into the formal reasoning loop. H-Logic remains the authority for derivation, but it can consult a latent model when choosing what to explore.

Potential guidance services include:

- premise selection;
- rule selection;
- branch ranking;
- join and traversal planning;
- conjecture generation;
- relation proposal;
- contradiction or anomaly prioritization.

The safe default is a *reordering heuristic*. The reasoner first builds the same complete candidate set that it would use without a model. Neural scores then affect ordering, not membership.

```
Candidates := BuildCompleteCandidateSet(Goal)
Scores     := Intuition.Rank(Goal, Candidates, Context)
Ordered    := ReorderWithoutRemoving(Candidates, Scores)
Result     := ExploreFormally(Ordered)
```

Listing 67.3: Completeness-preserving guidance

The critical guarantee concerns validity, not guaranteed completion within a finite operational budget:

> A neural intuition must not change what constitutes a valid proof. Under a finite timeout, depth limit or node budget, however, it may change which valid proofs are reached before the budget is exhausted.

This distinction is operationally important. A low-scored candidate is postponed rather than discarded in safe mode, but a poor ordering can still cause a proof that was previously reached within budget to time out. Resource exhaustion must therefore be reported as an explicit operational outcome, never as formal falsity. The safe claim is that every proof produced remains valid and that, when both complete searches terminate, their formal conclusions are equivalent.

The guarantee also assumes that the reordered operation is semantically order-insensitive. Before guidance is enabled, H-Logic features such as negation by failure, cut-like operators, first-result semantics or first-N result limits must be identified. Where observable semantics depend on order, guidance must be disabled, constrained to a semantically neutral region, or declared as an approximate operational policy.

Approximate pruning can be supported later, but it must be explicitly declared and its results must carry an approximate-search qualification.

67.4 Why the Hypergraph Changes the Integration Problem

Many neuro-symbolic systems must bridge two incompatible worlds: a prover manipulates terms and predicates, while a model consumes anonymous vectors. logiCells starts from a different

position because formal reasoning and latent projection are derived from the same conceptual hypergraph.

A conceptual tensor can preserve:

- the stable identity of each concept;
- the formal type of each relation;
- the ordered roles of a polyadic relation;
- the analysis associated with each tensor axis;
- the version of the topology and index mapping;
- the provenance of projected values.

The bridge is therefore not merely a serialization step. It is an explicit mapping between symbolic identities and numerical positions. This mapping is the foundation for both governed materialization and neural guidance of formal search.

67.5 Hypergraphs, Not Merely Binary Graphs

A standard graph neural network often assumes binary edges. A logiCells relation may be polyadic and role-sensitive. Flattening such a relation into a clique or a star can erase arity, role names and relation identity.

Consider a transaction relation with buyer, seller, product, channel and date roles. A role-aware conceptual representation should preserve the incidence structure:

```
participant
    <-> participant role
    <-> hyperrelation instance
    <-> participant role
    <-> another participant
```

Listing 67.4: Role-aware hypergraph incidence

A future conceptual tensor may therefore use three or more axes:

```
inputs:
  - name: transactionIncidence
    kind: conceptual
    formalType: Transaction

    axes:
      - name: relation
        analysis: relationIdentity

      - name: role
        analysis: participantRole

      - name: participant
        analysis: participantIdentity
```

Listing 67.5: A rank-three role-aware conceptual projection

This is a natural area for native logiCells operations. Generic dense runtimes can execute linear algebra after projection, but they do not know the semantics of hyperarities, formal roles or incremental conceptual identity.

67.6 Design Commitments

The following commitments define the intended position of the platform.

67.6.1 Neural Performance Must Not Be Confused with Formal Correctness

In safe guidance mode, correctness and performance are evaluated separately. For unbounded searches that both terminate, the formal conclusions must be equivalent. Under bounded resources, a result may instead be *resource exhausted*; that status is neither proof nor refutation.

```
if both complete searches terminate:
    guided conclusions = unguided conclusions

under a finite budget:
    every produced proof remains formally valid
    resource exhaustion remains explicit

on a reference workload:
    aggregate guided cost < aggregate unguided cost
```

Listing 67.6: Correctness and performance are evaluated separately

Performance is an aggregate benchmark property, not a per-proof semantic assertion. Some goals are already trivial, some receive no useful guidance and some may become slower. A backend or model is accepted only when it preserves the semantic contract and improves a declared workload without hiding regressions behind timeout or failure statuses.

67.6.2 Every Intuition Is Versioned

A score is meaningful only relative to:

- a model artifact version;
- a hypergraph state or snapshot;
- a projection definition;
- an index mapping;
- a latent-space version;
- an evaluation time.

A stale intuition must never be used silently. The runtime must be able to wait for an updated view, fall back to symbolic-only reasoning, or explicitly allow bounded staleness.

67.6.3 Proposals Are Not Conclusions

Link predictions, conjectures and missing-premise suggestions are represented as proposed symbolic structures with confidence and provenance. They may then be proved, refuted, retained for human review or allowed to expire.

67.6.4 The Native Runtime Remains Focused

logiCells should retain native responsibility for operations that depend on conceptual identity and hypergraph semantics:

- stable and local index mappings;
- sparse conceptual projections;
- incremental invalidation after hypergraph mutations;
- role-aware hypergraph message passing;

- conversion between proposals and governed conceptual observations.

Generic dense computation and large language-model inference can use more than one runtime path. logiCells offers a provider-neutral LLM service boundary and a resident inference integration capable of reading GGUF model assets and can execute with platform acceleration such as MLX on supported Apple Silicon deployments. The resident path includes persistent model state and token-by-token generation infrastructure, including a KV cache. External engines such as llama.cpp remain useful reference or provider paths rather than the sole implementation model.

The public contract is the capability and lifecycle boundary, not the internal implementation type. Application code should select the supported inference backend for its deployment and must not assume that every backend is available on every host.

67.7 Architectural Scope and Explicit Non-Goals

logiCells does not currently compile H-Logic rules into neural weights or embed the complete formal semantics inside a differentiable architecture. That stronger `Neuro[Symbolic]` direction is a different design choice. It can be valuable for constrained learning, but it couples validity, training behavior and model replacement more tightly than the governed architecture adopted here.

The present scope is deliberate:

- symbolic structures remain inspectable and independently executable;
- neural components provide scores, representations and proposals;
- formal proof obligations remain in H-Logic and the conceptual runtime;
- approximate policies are explicit rather than hidden in an optimizer;
- model replacement does not redefine the meaning of an accepted proof.

This choice favors auditability, operational replacement and human supervision. Differentiable constraints may be added later as another composition pattern, but they will not silently replace the governed contract.

67.8 Terminology and Primary References

This guide uses *governed compositional neuro-symbolic integration* for the application-boundary pattern and *Symbolic[Neuro] deduction guidance* for the reasoner-internal pattern. These terms are used consistently throughout the remaining chapters.

The positioning is informed by:

- Henry A. Kautz, “The Third AI Summer: AAAI Robert S. Engelmore Memorial Lecture,” *AI Magazine*, volume 43, number 1, pages 93–104, 2022, based on the AAAI 2020 Engelmore lecture;
- Frank van Harmelen and Annette ten Teije, “A Boxology of Design Patterns for Hybrid Learning and Reasoning Systems,” *Journal of Web Engineering*, 2019;
- Michael van Bekkum, Maaike de Boer, Frank van Harmelen, André Meyer-Vitali and Annette ten Teije, “Modular Design Patterns for Hybrid Learning and Reasoning Systems: A Taxonomy, Patterns and Use Cases,” *Applied Intelligence*, volume 51, number 9, pages 6528–6546, 2021.

The taxonomy is used as a map of architectural compositions, not as a claim that one box is universally more mature than another.

67.9 Current Status and Intended Evolution

The current implementation has two different maturity levels that should not be conflated. The *symbolic gate around AI proposals* is implemented and covered by executable tests. The deeper `Symbolic[Neuro]` search-guidance architecture is still being developed incrementally.

The status must remain explicit throughout the guide:

Implemented and tested – governed proposal gate H-Logic candidates carry provenance/origin, parser validation is detached from activation, rejected candidates cannot be committed, and accepted candidates pass through an explicit transaction/commit lifecycle. The current tests include LLM-origin candidates.

Implemented and tested – neural/tensor foundation Declarative `.ia` loading, dense and conceptual inputs, selected layers, supervised training, graph convolution and GraphSAGE over conceptual topology.

Being designed Deduction traces, intuition interfaces, versioned latent views and completeness-preserving candidate reordering.

Planned Premise and rule selection, query planning, role-aware hypergraph message passing, incremental latent indexes and governed conjecture generation.

This distinction is important: "the LLM proposes and the symbolic runtime decides" is already an executable governance contract. Neural guidance of the internal proof search is a separate, less mature capability.

67.10 Positioning Summary

logiCells does not claim that all neuro-symbolic integration must be differentiable. It deliberately supports two complementary forms:

Governed composition Neural computation produces observations and proposals whose semantic effects remain controlled by the conceptual runtime.

Symbolic[Neuro] Formal reasoning calls neural intuition to prioritize search while preserving symbolic authority over conclusions.

The resulting promise is concise:

> The neural component may determine where the reasoner looks first, but never what constitutes a valid proof.

Under bounded resources, guidance may still affect whether a proof is reached in time. Resource exhaustion therefore remains explicit and is never interpreted as formal failure.

Chapter 68

Agents Operating on Conceptual Models

An agent in logiCells should operate on the conceptual model rather than on disconnected technical artifacts. It can inspect concepts, follow relations, call published actions, consume messages, produce events and coordinate with process objects.

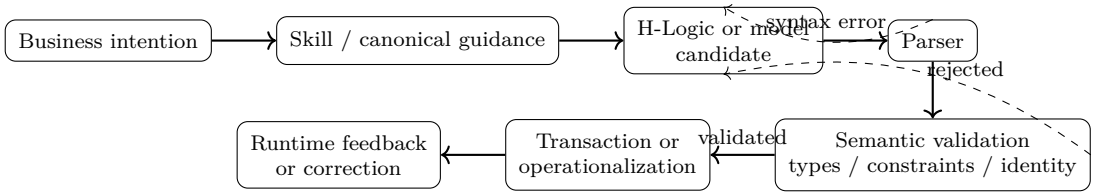


Figure 68.1: Agentic generation is a governed correction loop: the Skill guides generation, while parser and semantic validation decide acceptance.

68.1 Purpose

This chapter describes how agents access a logiCells application through published conceptual capabilities. The agent may be implemented by a third-party coordinator and may cross a publication boundary such as MCP. The protocol changes the access mechanism; it does not redefine the conceptual capability.

Resident agent execution is available through the Agent runtime and CLI. The conceptual rules in this chapter are complemented by the executable session workflow in Chapter 69, “Agent Runtime, Goals and Persistent Sessions”.

68.2 Core Ideas

Agents operate through explicit model elements:

Concepts define the objects and capabilities available to the agent.

Actions provide callable operations.

Published actions provide stable execution handles.

Events allow agents to react to changes.

Messages allow asynchronous coordination.

Processes organize long-running work.

68.3 Publication Boundary and MCP

An external agent should see a controlled projection of the application rather than an independent technical API that duplicates business semantics. A publication layer can expose selected conceptual resources and published actions to an agent coordinator. MCP is one possible protocol for that boundary.

The current platform supports MCP in both directions. Tooling contains a governed MCP *client/operationalization* path with discovery, explicit read-only/effectful allow-lists, policy checks, schema-digest checks and tests around tool execution. Discovery is explicitly not treated as authorization.

The CLI also exposes MCP compilation for applications and packages. A compiled application MCP surface can be listed in the application's `.lgc` Agents configuration, where the Agents runtime starts the configured MCP server and makes the published application boundary available to external MCP clients. See Chapter 61, "Publishing Applications through OpenAPI and MCP" for the developer workflow.

Publication and server-side authorization are separate responsibilities. For any effectful MCP operation, check the authentication, `PolicyContext`, `EvidenceRequirements` and `PendingConfirmation` behavior of the deployed inbound adapter. The required action-governance principles are described in Chapter 70, "Governed Agent Execution Contract".

The conceptual contract remains model-driven:

```
logiCells model
    -> published conceptual capabilities
    -> publication adapter / MCP boundary
    -> external agent
```

The publication layer should preserve declared names, typed parameters, authorization scope and validation rules. It should not invent a parallel ontology for the same business operations.

68.4 Using VIEW Interaction Contracts without Rendering a Form

An agent can consume the same VIEW interaction contract as a native or web client without materializing a visual form. The contract provides a structured description of the current data context, interaction functions, MODEL-behind behavior, available actions and event relationships.

The agent can therefore follow this pattern:

```
load interaction contract
-> inspect available/current values
-> determine which values are still required for the intended action
-> present useful state to the user
-> ask only for missing information
-> invoke the declared published/model action
-> observe result/event and continue the interaction
```

Listing 68.1: Agentic enactment of an interaction contract

This keeps the agent aligned with the same application interaction semantics as human-facing clients. The contract is not an instruction to synthesize a fake form; it is a channel-independent interaction plan. Authorization, policy and confirmation requirements still come from the governed execution boundary described later in this guide.

68.5 Agent Invocation of Published Actions

A published action can be called by an agent in the same way it can be exposed through a web service. The difference is the access mechanism, not the conceptual capability. This allows the same action to be reused by a user interface, an API, a process and an automation agent.

68.6 Agent and Process Coordination

When an agent participates in a multi-step workflow, the process object should remain the coordination authority. The agent may execute a step, validate data, call an external service or emit a result event, but the process state records where the operation is.

An external coordinator should therefore avoid reconstructing an application workflow when the process is already modeled in logiCells. It can start or advance the process through published capabilities and observe declared state or events.

68.7 Agent and Messaging

For asynchronous work, the agent can listen to messages or events and respond later. This is useful when processing may take time or require an external system. The response can be modeled as a message, event or action invocation.

68.8 Human-in-the-Loop Automation

When automation requires a human decision, represent the pause in the conceptual process rather than hiding it inside agent code. The process should expose the pending state, the relevant view or published action, and the event that resumes the workflow. This keeps the decision observable and auditable.

68.9 AI Agents and the Symbolic Gate

An AI agent may inspect conceptual state, propose actions, summarize events, classify candidates or suggest model changes. It remains a proposer, not an authority. Generated H-Logic should be treated as a candidate and passed through the official parser and semantic validation path before it can participate in a transaction or operational execution.

When an executable validator is available to an AI Skill, the preferred loop is *propose* → *validate* → *correct*. A bounded retry policy is preferable to an unbounded self-repair loop: after a small number of unsuccessful corrections, return the last diagnostics and the unresolved business question to the developer.

68.10 Design Rules

Agents should use the same published capabilities, process states, events and messages as other runtime participants. Do not duplicate a business capability as an agent-specific function when a published action already exists. Treat MCP or another agent protocol as a publication adapter, not as a second business model. Long-running coordination belongs in process models. Critical operations must remain auditable through state changes, events and logs.

Chapter 69

Agent Runtime, Goals and Persistent Sessions

The `cell` CLI exposes the Agent runtime directly. Developers can run a task in one command or work with a persistent session whose lifecycle is explicit: open the session, submit a goal, inspect pending confirmations, confirm or reject an operation, inspect a snapshot, evaluate the session, and close it.

This chapter documents that executable surface. It does not depend on a particular user interface or commercial workflow.

69.1 One-Shot Agent Execution

The direct entry point is:

```
cell agent run <task> [--agent-id <id>] [--subject-id <id>]
              [--purpose <name>] [--profile <name>]
              [--knowledge-source <path>]
              [--knowledge-mode source|project]
              [--knowledge-answer deterministic|verbalized]
              [--knowledge-query <hlogic>]
              [--knowledge-nested-query <hlogic>]
              [--max-turns <n>]
              [--max-provider-iterations <n>]
              [--max-iterations <n>]
              [--interactive]
              [--on-confirmation fail|reject]
```

Use this form when a task can be executed as one CLI interaction and the caller does not need to manage a persistent session explicitly.

The identity/context options are distinct:

`-agent-id` Selects the agent identity when more than one agent definition is available.

`-subject-id` Provides the subject identity used by the execution context.

`-purpose` Supplies a named purpose to the runtime context.

`-profile` Selects an execution/profile configuration.

The iteration limits provide developer-controlled bounds around provider and runtime loops. Use them in tests and automation so an unexpected provider behavior cannot turn into an unbounded run.

69.2 Knowledge Inputs and H-Logic Queries

Agent execution can receive an explicit knowledge source and can query that source using H-Logic:

```
--knowledge-source <path>
--knowledge-mode source|project
--knowledge-answer deterministic|verbalized
--knowledge-query <hlogic>
--knowledge-nested-query <hlogic>
```

`source` mode addresses a supplied knowledge source directly; `project` mode uses a project-oriented knowledge boundary. The answer mode separates deterministic output from a verbalized form. H-Logic query arguments make the knowledge request explicit instead of leaving all context selection to free-form provider prompting.

For debugging, first validate the same H-Logic and semantic structures with the dedicated semantic commands when possible. Agent behavior is easier to interpret when the underlying query is known to be valid independently of the provider.

69.3 Canonical Agent Runtime Terms

The developer documentation uses the following capability names consistently across the CLI, runtime specification and capability matrix:

Agent Session / Goal The persistent session and its submitted objective, exposed by `cell agent session open` and `cell agent session goal`.

KnowledgeQuery The explicit knowledge-query capability surfaced by `-knowledge-query` and `-knowledge-nested-query`; the CLI option name is syntax, while KnowledgeQuery is the capability name.

PendingConfirmation A runtime state in which a proposed effectful operation is not yet dispatched and can be inspected through `cell agent session pending`.

Confirmation Broker The runtime responsibility that resolves a pending request through confirm or reject and resumes the owning session. The public CLI surfaces the lifecycle even when the implementation is behind a runtime service boundary.

PolicyContext The authorization context used to decide whether an operation is allowed, denied, or requires confirmation. Exact configuration syntax is runtime/version specific and must not be inferred from the CLI alone.

EvidenceRequirements Evidence conditions that may be required by policy before an operation can be authorized. Their concrete declaration syntax must be documented from the authoritative runtime schema before a copyable example is published.

These names describe capabilities, not command spelling. They are used in the governed-execution contract in Chapter 70, “Governed Agent Execution Contract”.

69.4 Persistent Agent Sessions

Persistent sessions expose the runtime state machine directly:

```
cell agent session open [--agent-id <id>]
cell agent session goal <session-id> <goal> [options]
cell agent session pending <session-id>
cell agent session confirm <session-id> <request-id>
```

```
cell agent session reject <session-id> <request-id>
cell agent session snapshot <session-id>
cell agent session evaluate <session-id>
cell agent session close <session-id>
```

Agent session commands require a persistent `-tooling-endpoint`. A transient STDIO Tooling process cannot preserve the session state between independent CLI commands.

A normal development sequence is:

```
cell --tooling-endpoint <url> agent session open
# record the returned session id

cell --tooling-endpoint <url> agent session goal <session-id> \
  "<goal>" --max-iterations 8

cell --tooling-endpoint <url> agent session pending <session-id>
# inspect a request id when confirmation is required

cell --tooling-endpoint <url> agent session confirm <session-id> <request-id>
# or:
cell --tooling-endpoint <url> agent session reject <session-id> <request-id>

cell --tooling-endpoint <url> agent session snapshot <session-id>
cell --tooling-endpoint <url> agent session evaluate <session-id>
cell --tooling-endpoint <url> agent session close <session-id>
```

Use `-json` when a script needs stable machine-readable results rather than human-oriented output.

69.5 Goals

A session goal is the task submitted to an already-open agent session:

```
cell agent session goal <session-id> <goal>
     [--agent-id <id>] [--subject-id <id>]
     [--purpose <name>] [--profile <name>]
     [--max-iterations <n>]
     [--knowledge-source <path>]
     [--knowledge-mode source|project]
```

A goal is not an authorization grant. It gives the agent an objective inside the runtime context. Available capabilities, policy checks, knowledge constraints and confirmation requirements still govern what can actually execute.

69.6 Pending Confirmations

Effectful work may suspend in a pending-confirmation state instead of executing immediately. The CLI makes this state inspectable:

```
cell agent session pending <session-id>
```

The resulting request identifier is then used explicitly:

```
cell agent session confirm <session-id> <request-id>
cell agent session reject <session-id> <request-id>
```

The important programming model is that proposal and authority remain separate. A provider can propose an operation; the runtime can hold it pending; another participant can accept or reject it; and the session remains the container for subsequent execution and continuation.

For one-shot execution, `-on-confirmation fail|reject` defines what the CLI should do if confirmation is required but no interactive confirmation flow is being maintained.

69.7 Snapshots and Evaluation

`snapshot` and `evaluate` are separate developer surfaces:

`snapshot` Inspects the current session state for diagnostics, tests or orchestration.

`evaluate` Requests evaluation of the session using the runtime's exposed session evaluation operation.

Do not substitute provider narration for session inspection. When debugging a multi-turn run, use the explicit session surfaces to establish what the runtime recorded.

69.8 MCP in Agent Execution

The Agent runtime can participate in MCP in both directions described in Chapter 61, "Publishing Applications through OpenAPI and MCP".

An external agent can connect to a logiCells MCP publication surface. Conversely, a logiCells agent can consume tools from configured external MCP servers. On the client path, discovery does not imply permission to execute. Read-only/effectful classification, policy and confirmation can remain independent stages of tool operationalization.

This distinction matters when diagnosing a tool that appears in discovery but is not executable in the current context: the publication contract, the catalog/policy decision and the pending-confirmation state are separate pieces of runtime state.

69.9 Local Providers

Provider settings are release-specific. Obtain a copyable `.lgc` provider configuration from the version-matched starter or official configuration schema; avoid copying an unrelated deployment profile. After configuration, exercise the provider through the same `agent run` or persistent-session commands.

Provider interchangeability does not change the conceptual execution contract: Goal, Knowledge, policy, confirmation, action/tool execution and recorded session state remain runtime concerns.

69.10 Testing an Agent Scenario

A developer-facing regression scenario should make each observable boundary explicit:

1. Start or connect to the persistent Tooling endpoint.
2. Open a session.
3. Submit a goal with bounded iterations.
4. Supply a known knowledge source when the scenario depends on knowledge.
5. Inspect pending requests rather than assuming an effectful tool executed.
6. Exercise both confirm and reject paths where applicable.
7. Capture a session snapshot.
8. Evaluate and close the session.
9. Repeat with `-json` in automation so assertions use the stable result envelope.

The test should also verify negative behavior: rejection must not be treated as successful execution, and a capability that is only discovered must not be assumed to be authorized.

69.11 Typed External Effects and Their Evidence

An outbound MCP operationalization may transform a provider response into typed knowledge bindings and supporting evidence. Preserve the distinction between the external tool's raw response, the verified business receipt and the context presented to an agent. Where the deployed runtime offers these types, inspect the action result and its provenance before the next step. A returned result is not by itself permission to retain it as shared or persistent knowledge.

For operations with side effects, check the action's required permissions, submitted arguments and confirmation request in the same runtime configuration that will execute the operation.

Chapter 70

Governed Agent Execution Contract

This chapter is a developer specification for the authority boundary shared by resident agents, inbound protocol calls and effectful tool execution. It separates normative behavior from availability claims. A requirement stated here is not evidence that every protocol path already implements it; the capability matrix records what is established by the supplied release evidence.

70.1 Authority Model

The runtime must keep publication, discovery, proposal, authorization, confirmation and execution distinct. The following rules are normative:

- Publication is not authorization.
- Discovery is not authorization.
- Provider output is a proposal, not an execution grant.
- An effectful operation must not be dispatched before all required policy and confirmation conditions are satisfied.
- Authorization must be evaluated against the execution context used for the actual dispatch.

A useful conceptual state sequence is:

```
existent -> published -> discoverable -> proposed
         -> policy evaluation
         -> allow | deny | pending confirmation
         -> confirmed (when required)
         -> revalidated
         -> dispatched once
         -> recorded result / audit
```

70.2 Protocol-Neutral Execution Context

Governance should attach to the action and execution context rather than to a specific publication protocol. The logical context for a governed operation includes, when available:

- caller or agent identity;

- subject identity;
- purpose;
- owning session;
- requested action/tool and typed arguments;
- KnowledgeQuery result/context;
- evidence used to satisfy EvidenceRequirements;
- PolicyContext;
- confirmation state;
- dispatch result and trace/audit identity.

The CLI exposes several of these directly through `-agent-id`, `-subject-id`, `-purpose`, knowledge options, session identifiers and pending request identifiers.

70.3 Governed Action Metadata

The logical action contract must be able to represent the following concerns even if the concrete YAML/XML schema uses different names:

Publication Whether the action is externally exposed.

Effect classification Whether execution may change state or produce an external side effect.

Policy binding Which policy or PolicyContext governs the call.

Confirmation requirement Whether confirmation is never required, policy-dependent, or mandatory.

Evidence requirements Which evidence must be present before authorization.

Audit identity The stable identity used to record the requested operation and its outcome.

The concrete declaration syntax is intentionally not invented in this specification. It must be copied from the authoritative model/configuration schema and a validated starter before it is presented as a developer listing.

70.4 PendingConfirmation Semantics

When confirmation is required, the runtime enters a PendingConfirmation state instead of dispatching the operation.

70.4.1 Reject

A reject operation:

- MUST NOT dispatch the pending operation;
- MUST resolve the referenced pending request;
- MUST preserve the owning session unless a separate session failure occurs;
- MUST make the rejection observable to diagnostics/audit surfaces.

70.4.2 Confirm

A confirm operation:

- MUST identify one specific pending request;
- MUST NOT silently authorize a materially different argument payload;
- MUST revalidate any runtime conditions that can change between proposal and dispatch, or otherwise guarantee they remain valid;
- MUST authorize at most one dispatch for that confirmation;
- MUST record the resulting execution outcome so the session can resume from the actual result rather than from provider narration.

A replayed or duplicated confirmation identifier MUST NOT cause a second dispatch of the same one-shot operation.

70.5 Outbound MCP Tool Execution

For logiCells acting as an MCP client, the governed sequence is conceptually:

```
Agent Goal
 -> external MCP discovery
 -> Catalog / capability selection
 -> PolicyContext evaluation
 -> read-only dispatch OR PendingConfirmation
 -> confirm/reject when required
 -> external MCP call at most once
 -> result returned to the same session
 -> snapshot/audit
```

A discovered tool that fails policy, evidence or confirmation checks must remain non-executable in that context.

70.6 Inbound MCP Published-Action Execution

For an external MCP client calling a logiCells-published action, the target governed sequence is:

```
external MCP client
 -> MCP server
 -> resolve published action
 -> validate typed arguments
 -> construct execution context
 -> PolicyContext / EvidenceRequirements
 -> allow | deny | PendingConfirmation
 -> confirm/reject when required
 -> revalidate
 -> dispatch at most once
 -> result + audit
```

Publishing an application or package provides a protocol endpoint, not an authorization policy. Check the behavior of effectful inbound calls against the requirements above for the adapter and runtime actually deployed; do not infer confirmation or one-shot semantics from publication alone.

70.7 OpenAPI and Other Publication Adapters

The same principle applies to OpenAPI or future publication adapters. If governance is part of the action/runtime contract, protocol projection should not weaken it. An adapter may expose different transport semantics, but it should not bypass action-level policy, evidence or confirmation requirements.

70.8 Required Regression Scenarios

A release that claims governed execution should include versioned tests for the relevant path:

1. allowed read-only operation executes without an unnecessary confirmation;
2. denied operation produces no dispatch;
3. effectful operation that requires confirmation enters PendingConfirmation;
4. reject produces no dispatch and the session remains usable;
5. confirm causes exactly one dispatch;
6. replay of the confirmation does not dispatch again;
7. mutation of protected arguments between proposal and dispatch is rejected or revalidated;
8. KnowledgeQuery/evidence requirements fail closed when required evidence is missing;
9. result and rejection paths remain observable in the session snapshot/audit surface;
10. inbound MCP and resident Agent paths satisfy the same action-level authority rules when they target the same governed action.

70.9 Developer Quickstart Acceptance Criterion

The canonical starter for this contract should let a developer reproduce the following without internal repository paths:

```
published action
 -> configured policy/evidence/confirmation rule
 -> provider + persistent Tooling endpoint
 -> session open
 -> Goal + KnowledgeQuery
 -> pending request
 -> reject => prove no effect
 -> new request
 -> confirm => one effect
 -> session resumes
 -> snapshot/audit inspection
```

A second exercise should expose the same governed action through MCP and prove whether the inbound path enters the same authority boundary. The starter configuration itself is part of the developer contract and must be version-controlled with the distribution.

70.10 Deployment Checks for Published Actions

Before exposing an action through MCP, OpenAPI or another protocol, verify that the deployed adapter enforces the application's authentication, authorization and confirmation requirements on the actual inbound path. Publish only the minimum necessary operations. Use a separate test environment to exercise denied, repeated and cancelled requests as well as the successful case. The availability of a publishing adapter does not automatically establish the behavior of a complete governed workflow.

70.11 Developer Assistance and Knowledge Boundaries

Keep application operations separate from workspace-editing privileges. An agent that can run a business action must not automatically gain permission to modify source models. Likewise, contextual evidence returned by a tool is not necessarily approved for persistent, shared knowledge. Define explicit scopes, approval rules and retention requirements before enabling either behavior.

Chapter 71

Human-in-the-Loop Conceptual Workflows

Many workflows cannot be fully automated. They require human validation, data entry, correction, approval or interpretation. In logiCells, human participation should be modeled as part of the workflow rather than hidden inside screens.

71.1 Human Tasks as Process Steps

A human task can be represented by an action, a view, a state or an activity object. The process stores where the user is expected to intervene and what result is required.

71.2 Views as Task Interfaces

The view used for a human task is a projection of the process or business object. It should not own the workflow. It should collect or display the information needed for the current state.

71.3 AI Assistance

An AI assistant can summarize the current state, propose a decision, detect missing information or prepare a response. The human remains responsible for the validation when the process requires it.

71.4 Design Rules

Make human tasks explicit. Record the process state before and after intervention. Use events to mark completion. Keep the view as an interface, not as the owner of the workflow. Allow agents to assist without bypassing validation rules.

71.5 Summary

Human-in-the-loop workflows are more maintainable when human intervention is modeled as part of the process, not as an implicit UI side effect.

Chapter 72

Explainability, Traceability and Governance

Executable conceptual models must be explainable. A user, auditor, developer or agent should be able to understand why an object exists, which process changed it, which action was called, and which event was emitted.

72.1 Explainability from Structure

logiCells improves explainability because the model names the concepts, roles, actions and processes. An explanation can refer to model elements rather than to anonymous code paths.

72.2 Traceability from Events and Logs

Events and logs complement each other. Events describe meaningful changes. Logs record execution details. A good trace links the object, action, event, process state and user or agent context.

72.3 Governance of Published Capabilities

Published actions must be governed. The model should make clear which capabilities are exposed, which parameters are accepted, which roles may call them and which side effects they create.

72.4 AI Governance

When AI assistants operate on conceptual models, governance becomes even more important. The assistant should operate through explicit actions and authorized contexts. It should not bypass model constraints or invent capabilities outside the model.

72.5 Design Rules

Name important actions and events clearly. Keep process states explicit. Log state transitions and service calls. Avoid hidden side effects. Treat published actions as governed contracts.

72.6 Summary

Explainability is not an afterthought. It is a consequence of modeling concepts, actions, events, states and publication boundaries explicitly.

Part X

Conceptual Engineering Practices

Chapter 73

Testing Conceptual Models

Testing a logiCells application means testing more than code. It means testing concepts, relations, constraints, actions, events, processes, service publication and view projections. A conceptual model is executable, so it must be validated as an executable artifact.

73.1 Purpose

This chapter introduces testing practices for declarative conceptual models. The goal is to make model evolution safe and to detect inconsistencies before they appear in runtime behavior. Application tests should be expressed as close as possible to the same conceptual surface used by the application itself.

73.2 What to Test

A complete test strategy should cover:

- loading of model files;
- concept inheritance and implications;
- hypergraph and database facets;
- role paths and linked object projections;
- collection providers and joins;
- action signatures and execution;
- published action access;
- event routing;
- process state transitions;
- view bindings and datasets;
- service exposure and request mapping.

73.3 Structure a Declarative Test Package

Keep application tests in a dedicated package when they are not required by the production application. A typical module can therefore separate its production model and its tests:

```
my-module/
  packages/
    MyModule.package.yaml
    MyModule.Tests.package.yaml
  meta/
    models/
      MyModule/
        _.manifest.yaml
        Customer.model.yaml
      MyModuleTests/
        _.manifest.yaml
        CustomerModelTests.model.yaml
        CustomerActionTests.model.yaml
  src/
  tests/
```

Listing 73.1: Recommended module and test layout

The test package follows the normal package-loading rules. Its schema directory must make the test manifest reachable, and the application used for testing must load both the application package and the test package. Merely placing a test model in the source tree does not make it part of the test run.

A test package descriptor follows the same shape as an application package:

```yaml
package:
  items:
    - MyModuleTests:
        caption: '#MyModuleTests'
        enabled: true
  files:
    schemaDirectories:
      schemaPath:
        value: '[Modules]\my-module\meta\models'
```

Listing 73.2: Test package descriptor

Its manifest declares the test-suite models normally:

```yaml
package:
  objectsContainer: CustomObjectsContainer
  plugginCaption: My Module Tests
  uri: http://www.logicells.net/schema/MyModuleTests
  classes:
    - CustomerModelTests:
        classId: myapp#customer_model_tests
        type: Entity
    - CustomerActionTests:
        classId: myapp#customer_action_tests
        type: Entity
```

Listing 73.3: Test package manifest

73.4 Declare a Test Suite as a Model

A declarative test suite is an ordinary model whose `Entity` concept implies `lgc#unit_test`. This identifies the model as a unit-test suite while keeping the tests inside the declarative application model.

```yaml
class:
  name: CustomerModelTests
  classId: myapp#customer_model_tests
```

```
  className: CustomClassItem
  currentGeneratorName: MyModuleTests.CustomerModel
  type: Entity

  generators:
    - MyModuleTests.CustomerModel:
        class: GUIDStringGeneratorInfo
        params: ''

  object:
    className: CustomObjectItem
    objectName: CustomerModelTests

  concepts:
    - Entity:
        constraints:
          - implies: lgc#unit_test
```

Listing 73.4: Minimal declarative test suite

Do not replace the `Entity` concept with only `type: Entity`. The suite must participate in the conceptual model as an Entity and explicitly imply `lgc#unit_test`.

73.5 One Test Case per Declared Action

Each Action declared directly by the test-suite model represents a test case. For a scripted application test, use `UnitTestScriptableObjectElementAction`. Give the Action a descriptive name that states the contract being protected.

```
actions:
  - action:
      name: Customer_Class_Is_Loaded
      class: UnitTestScriptableObjectElementAction
      type: ClassItem
      caption: '#Customer_Class_Is_Loaded'
      published: true
      scripts:
        - script:
            clientScript: true
            serverScript: false
            language: functionalP
            classes: Desktop
            cdata: |
              TargetClassItem := Self.ObjectsContainer.ClassItemByClassId('myapp#customer');
              Self.Assert(
                GUnitTest,
                (TargetClassItem <> nil),
                'Customer ClassItem is not loaded'
              );
```

Listing 73.5: A model-loading test case

The test Action can inspect the same conceptual runtime surface as other application Actions. This makes it suitable for checking whether classes are loaded, metadata is present, Actions are available, process states exist, or application behavior produces the expected result.

A single test case may contain several assertions when they protect one coherent contract. Prefer several small Actions over one large test Action that checks unrelated parts of the application.

73.6 Assertions in functionalP

Use `Self.Assert` with `GUnitTest` inside a unit-test Action:

```
Self.Assert(
```

```
  GUnitTest,
  condition,
  'Message explaining the expected contract'
);
```

Listing 73.6: Assertion pattern in functionalP

The message should identify the missing or broken application contract. Prefer messages such as `PromptDefinition.Version metadata missing` or `Start action is missing` to generic messages such as `Test failed`.

Tests should normally use public model identities, metadata and declared Actions. Avoid reproducing implementation algorithms inside a test script merely to compare two copies of the same logic.

73.7 Organize Suites by Contract

For a non-trivial module, split tests according to the conceptual boundary being protected. A useful pattern is:

```
MyModuleTests
  CustomerModelTests
    class loading
    required properties
    relations and implications

  CustomerActionTests
    action presence
    parameter contracts
    returned values

  CustomerProcessTests
    declared states
    Start / Continue / Retry / Cancel
    transition and feedback contracts
```

Listing 73.7: Example test-suite organization

For `logiCells.Agent`, the same pattern leads naturally to `AgentModelTests`, `AgentActionTests` and `AgentProcessTests`. The purpose of the split is readability and diagnosis, not framework layering.

73.8 Run Declarative Tests with sclgc

The application configuration used for the test run must load the module under test and its test package. Run the application with the `-t` option:

```
sclgc -c /path/to/application.application.ACC.lgc -t
```

Listing 73.8: Run all registered application tests

The long form is equivalent:

```
sclgc --config /path/to/MyApplication.ACC.lgc --test
```

Listing 73.9: Run tests with long parameter names

To write test results to a specific output directory, add `-to` or `-test-out`:

```
sclgc -c /path/to/MyApplication.ACC.lgc -t -to /path/to/test-results
```

Listing 73.10: Run tests and select the result directory

The declarative test model is platform-independent, while the concrete test-host executable and packaging are distribution concerns. Before putting this command in CI, verify that the target runtime package exposes the same test-host options on the selected platform.

73.9 Testing Published Actions

Published actions deserve special attention because they may be called through user interfaces, processes, agents or web services. A test should verify the action signature, parameter mapping, validation behavior, returned result and side effects.

Contract tests are especially useful for published Actions because the declaration is shared by several callers. Tests should protect stable parameter names, parameter directions, relevant data types and expected business effects.

73.10 Testing Processes

Process tests should protect observable workflow contracts rather than duplicate the implementation of the workflow. Useful checks include:

- required states are declared;
- entry and continuation Actions exist;
- retry and cancellation contracts remain available when required;
- feedback and error information remain observable;
- business validation happens before a final state or materialization when required.

When a process invokes an agent or external service, keep the process-state assertions deterministic. Model-output quality tests and external-provider availability tests belong in separate integration suites.

73.11 Testing Asynchronous Behavior

Asynchronous behavior should be tested through events and messages. The test should verify that an event is emitted, the right handler is triggered, the process state changes correctly and no hidden client-specific assumption is required.

73.12 Testing Views

A view test should not validate appearance first. It should validate binding: the input cursor, datasets, data sources, collection paths and action bindings. Concrete rendering belongs to the registered component implementation.

73.13 Regression and Model Evolution

When a model evolves, tests should protect public contracts. Published action signatures, class ids, role paths, process states and service mappings should not change accidentally. If they change intentionally, the change should be visible in tests and documentation.

A practical rule is to add or update a declarative test whenever a model change introduces a new stable application contract. This keeps examples, runtime behavior and application expectations aligned.

73.14 Summary

A declarative test suite is a model that implies `lgc#unit_test`. Each Action declared by the suite is a test case, scripted tests can use `Self.Assert(GUnitTest, ...)`, and the test package is loaded through the same package and manifest mechanisms as other application models. With the test package present in the application configuration, `sclgc -c <application.lgc> -t` runs the application tests.

Chapter 74

Debugging Declarative Applications

Debugging a declarative application means following model execution rather than only stepping through procedural code.

74.1 Debugging Path

Start from the concept. Check the class identifier, facets, role paths, collections, actions and events. Then inspect the runtime object and the current evaluation context.

74.2 Common Failures

Common failures include missing role paths, wrong class identifiers, unposted object changes, incorrect collection paths, unbound datasets, unpublished actions and events without listeners.

74.3 Logging

Logs should include model names. A log line such as `[RiskReviewProcess.Finalize]` is more useful than a generic message.

74.4 Debug the Model Before the Script

When behavior is unexpected, first check the declaration. Verify the class identifier, action owner, action type, event binding, collection path and view target. Many runtime issues come from an incorrect declarative path rather than from the script itself.

74.5 Debugging Order

A practical order is: model loading, concept resolution, object creation, collection provider, action binding, event routing, view cursor and persistence. This order follows the runtime dependency chain.

74.6 Use Trace Names

Log messages should include conceptual names: class identifier, object identifier, action name, event name and process state. This makes logs useful to developers, support teams and AI-assisted diagnostic tools.

74.7 Loading and Semantic Diagnostics

Several diagnostics identify different layers of the declarative pipeline. Treat them as clues about the failed layer rather than as generic semantic errors.

Model never appears in the compile log. Check application configuration, package activation, schema path and manifest membership.

There is no Entity record. The class may declare `type: Entity` without an explicit conceptual `Entity` record. Inspect `class/concepts`.

No polyadic map between two concepts. Check that both concepts were semantically registered and that their positive roles can be mapped. For ordinary entities, verify their implicit `Me` semantics before adding any explicit map.

Duplicate classId. Find every active package and manifest that declares the identifier. Use another identifier for another meaning; use explicit redefinition only for an intentional replacement.

Different source than expected. Inspect the resolved physical package/model path. A second checkout or historical package may be active even when the edited source file is correct.

For a newly created module, a practical first command-line smoke test on a distribution that provides the `sclgc` runtime host is:

```
1 sclgc -c <path-to-application.ACC.lgc>
```

Listing 74.1: Declarative runtime smoke test

Then read the log in order: application, package, manifest, model, collections, semantic initialization, runtime start. Stop at the first layer where the expected module disappears or a structural error is reported.

For command-line diagnostics, remember that the runtime accepts long and short parameter names. Typical examples are `-debug`/`-dbg`, `-log-level`/`-ll`, `-test`/`-t`, `-test-out`/`-to`, `-modules`/`-m` and `-services`/`-sg`. A parameter declared with a value consumes the following argument; a flag does not. Under strict parameter control, unknown parameters and missing required values are errors rather than silently ignored input.

74.8 Debugging a YAML-Embedded FunctionalP Action

An embedded FunctionalP block introduces a second location system: the physical YAML line and the line inside the compiled script AST. The interactive debugger operates on `ScriptIdentity` and AST lines; the editor should map both directions using parser-derived provenance instead of rewriting AST line numbers. An editor adapter with source mapping may project breakpoints and stack frames onto physical YAML locations, including pending, verified, unresolved and stale breakpoint states. Check this integration in the installed extension before relying on YAML-line breakpoints. Chapter 75 gives the source mapping, CLI baseline and test criteria.

74.9 Summary

The fastest debugging path follows conceptual structure: declaration, runtime object, action, event, view and service.

Chapter 75

Source-Level Debugging of FunctionalP in YAML Models

75.1 What Is Supported and What Is Targeted

The native debugger operates on compiled FunctionalP/AST programs. The `cell debug` CLI connects to the debug module loaded by a running `sdlgc --ast-debug` process. Debugging a FunctionalP block directly at its physical line in a `.model.yaml` or `.models.yaml` file additionally requires an editor adapter that supports source-to-script mapping. Check that the installed adapter exposes this capability before using embedded-YAML breakpoints.

75.2 Design-Time and Runtime Must Stay Separate

```
VS Code editor
  |-- ToolingClient -> logicells-tooling JSON-RPC
  |     MODEL / APPLICATION / SEMANTIC / SCRIPT / compile-plan
  |
  `-- Debug Adapter -> RuntimeDebugClient -> sdlgc --ast-debug
        processes / breakpoints / stack / locals / steps / REPL
```

Tooling owns source parsing, semantic resolution and compilation. The runtime debug engine owns stops, execution state and stepping. VS Code is a client of both authorities; neither client is an excuse to duplicate parsers or the FPL execution engine in TypeScript.

75.3 Source Provenance Is a First-Class Compilation Product

Every embedded script requiring source debugging needs a stable mapping of runtime identity to its physical document:

Source identity `SourcePath`, `SourceDocumentId`, `SourceUnitId` and optional business identity such as `Package::Class@Action`.

Location `ContentStartLine`, `ContentEndLine`, optional column, and source kind such as `yaml-literal`, `fp`, `xml` or generated code.

Execution identity `ScriptIdentity`, AST line, executable lines and the relevant compilation generation.

For a literal YAML scalar `|` with a deterministic physical-to-content relationship, the basic conversion is:

$$\text{AstLine} = \text{SourceLine} - \text{ContentStartLine} + 1, \qquad \text{SourceLine} = \text{ContentStartLine} + \text{AstLine} - 1.$$

Internal blank lines count. The AST retains local line numbers: modifying the FunctionalP compiler to fake document line numbers would break the separation of responsibilities. Folded YAML `>` does not automatically admit this simple mapping; reject it for source debugging or use an explicit parser-generated source map.

75.4 Breakpoint Identity and State

The canonical breakpoint key is `ScriptIdentity` combined with `AstLine`, not a bare file path or line. Two Actions in one YAML file may both have an AST instruction at line 1. The debugger interface in `logiCells.Debugger.WebPage.pas` accepts `scriptIdentity` on the examined breakpoint route, and the client engine exposes script identity on stack frames; this is a useful runtime prerequisite, not proof of a completed YAML SourceMapRegistry.

PENDING A known source location whose compiled script or executable line is not yet available.

VERIFIED The script identity and executable AST line have been resolved for the current generation.

UNRESOLVED The physical line belongs to no executable instruction or known embedded script.

STALE The old identity refers to a previous compilation generation and must be rebound.

Do not return a verified DAP breakpoint simply because a request was accepted. After an incremental recompile, invalidate stale script bindings and resolve against the newly compiled generation. A failed resolution must carry an explicit reason. Source registration should occur outside the inner runtime stop-check; do not parse YAML each time the debugger evaluates whether to stop.

75.5 DAP Source Projection and Runtime Commands

A `setBreakpoints` request on a YAML physical line resolves that line to the containing Action/script, computes AST line, obtains the current script identity, installs a runtime breakpoint and reports the actual resolution to VS Code. On stop, the runtime returns instruction and stack identities; the adapter projects them back to YAML source and physical lines. `step`, `step-in` and `step-out` follow the current frame and must switch source documents when execution crosses embedded scripts.

The existing CLI commands provide a useful independent debug baseline:

```
cell server start <app.lgc> --foreground
cell server status
cell script compile <script.fp> --output <script.ast>
cell debug break add <script.ast> <astLine>
cell debug run <script.ast>
cell debug processes
cell debug stack <processpid>
cell debug locals <processpid>
```

```
cell debug step <processpid>
cell debug step-in <processpid>
cell debug step-out <processpid>
cell debug continue <processpid>
```

The `processpid` returned by `debug run` identifies a logical runtime process, not the operating-system PID of `sdlgc`. The set of supported step, evaluation and termination operations depends on the installed debugger; verify them against the actual adapter. A valid response containing an empty locals list does not guarantee that local-variable inspection is configured.

75.6 Source Mapping and Incremental Compilation Meet at Identity

The compiler's normalized source digest, semantic/behavior signatures, produced AST index and source provenance serve different purposes. Reusing a physical signature does not authorize retaining a breakpoint tied to a previous executable AST. When a script or relevant dependency changes, version-aware source maps and breakpoint bindings must be invalidated. Hot reload may reuse the same provenance model in the future, but it is not a prerequisite for the first YAML source-debug vertical.

75.7 Acceptance Scenarios

1. Two embedded scripts share physical document and AST line numbers but stop independently by `ScriptIdentity`.
2. Breakpoint on a YAML literal block maps to the correct local instruction, including blank lines.
3. Unsupported folded scalars produce an explicit unmapped diagnostic.
4. Stack, step and nested call frames return to the correct YAML source and physical line.
5. Source recompile makes old breakpoints stale, then correctly re-resolves or reports not executable.
6. Tooling and runtime debug errors remain distinguishable; no automatic source mutation or hot reload is claimed.

These scenarios describe the behavior expected from an editor adapter with embedded-YAML source mapping. Verify physical source line resolution, breakpoint relocation after recompilation and the indicated breakpoint state with the installed version. Without that adapter, use the compiled-script debugger and command-line interfaces.

Chapter 76

Logging, Diagnostics and Traceability

Logging records execution. Diagnostics explain what failed. Traceability connects runtime activity back to the model.

76.1 Trace Units

Useful trace units include object identity, class identifier, action name, event name, process state, user or agent context, and publication channel.

76.2 Diagnostics

A diagnostic message should identify the model element involved. If a view cannot open, report the view name and path. If an action cannot execute, report the action name and missing parameter.

76.3 Traceability

Events and process states provide a higher-level trace than logs alone. They show what happened in business terms.

76.4 What to Log

Log conceptual facts, not only technical messages. A useful trace includes the concept, object identifier, action, event, process state and result.

76.5 Diagnostics for Events

For an event-driven application, diagnostics should show which event was emitted, which source produced it, which actions were bound, and whether each handler completed. This makes asynchronous execution auditable.

76.6 Traceability and Governance

Traceability supports governance because it connects a decision or state change to the model element that produced it. This is especially important when webservices, agents or AI assistants

call published actions.

76.7 Summary

Good diagnostics make the model observable. Logs should serve the conceptual architecture, not hide it.

Chapter 77

Performance, Profiling and Model Complexity

Declarative modeling does not remove performance concerns. It makes the performance path more explicit: model activation, hypergraph traversal, provider execution, dataset/cursor movement, path expressions, actions, events, serialization, service publication, remote calls and worker scheduling can all contribute to observed latency.

77.1 No Benchmark Claim in This Revision

This edition intentionally publishes **no throughput, latency or scale benchmark**. Benchmarking is a separate validation pass. The absence of numbers must not be replaced by qualitative claims such as “fast”, “low overhead” or “scales to large graphs” unless a reproducible benchmark supports them.

What this chapter can do now is define what should be measured and which design choices create avoidable work.

77.2 Measure the Whole Path

A useful measurement identifies the boundary being timed. Typical layers include:

- package/model load and runtime activation;
- H-Logic parse and query execution;
- hypergraph lookup, traversal and persistence;
- collection-provider query time and result cardinality;
- DataSet/DataCursor paging and refresh in rich clients;
- action execution and process transitions;
- event/message routing;
- serialization and service publication;
- ABI or RPC calls, including marshalling and write-back;
- local/resident AI inference, model load and per-request execution.

Do not optimize one layer merely because it is visible. Measure from the caller’s boundary, then split the path until the dominant cost is identified.

77.3 Model Complexity

Complexity grows with the number of active concepts, relations, rules, joins, events and publication surfaces, but raw counts are not sufficient. Fan-out, query selectivity, repeated traversal, refresh frequency and remote boundaries often matter more than the number of declarations.

Prefer models in which high-frequency paths are explicit and bounded. Rich conceptual structure is not itself a performance defect; accidental repeated work is.

77.4 Collections and Joins

Collections and joins should be meaningful and bounded. Apply filters and context before expensive composition when possible. Avoid refreshing broad joins on every client event. If a client needs only a page or projection, do not materialize the entire logical collection merely to discard most of it later.

77.5 H-Logic and Hypergraph Queries

Record query mode, scope and result cardinality when investigating H-Logic performance. Homomorphic matching, unification and resolving queries do not necessarily have the same search behavior. A benchmark must name the grammar/query mode and dataset shape it measures.

When a query is used in a hot path, prefer explicit anchors and selective patterns over unconstrained exploration. Preserve correctness first; do not replace a semantic query with a faster but weaker one without documenting the changed meaning.

77.6 Local versus Remote Boundaries

Remote chattiness can dominate otherwise cheap operations. Prefer coarse runtime operations over host-language loops that repeatedly fetch one node, one field or one relation across RPC. The same principle applies to ABI boundaries when repeated marshalling can be replaced by one higher-level operation.

For long-running work, use process state, background execution or messaging rather than blocking a user-interface or request thread.

77.7 Rich Clients

For View-based clients, measure cache hit/miss behavior, DataSet refreshes, paging, synchronization, and the number of events escalated beyond the client. A local visual event should not become a server operation unless the model requires it. The companion Rich Client guide owns the detailed client-side patterns.

77.8 AI Runtime Measurements

When benchmarking local or resident AI, separate:

- model load/startup;
- first-token latency;
- steady-state token generation;
- context/KV-cache reset;

- memory footprint;
- backend and host platform.

A comparison is meaningful only when model, quantization, context, generation options and hardware are fixed.

77.9 Benchmark Publication Contract

A future benchmark published by logiCells should include at least:

- framework/runtime version and commit/build identifier;
- operating system, architecture and hardware;
- dataset/model size and shape;
- exact operation or query;
- warm-up policy and number of runs;
- median plus a dispersion measure, not only a best case;
- whether persistence, network, cache and synchronization are enabled;
- source or fixture needed to reproduce the measurement.

Until such results are published, this guide documents performance mechanisms and measurement practice, not performance claims.

Chapter 78

Extension Patterns and Common Pitfalls

Extensions are useful when they add a reusable capability without weakening the conceptual contract. The safest extensions preserve a clear distinction between business meaning, runtime contract, implementation and publication.

78.1 Useful Extension Patterns

Prefer registered capabilities, explicit adapters, published actions, declarative services and replaceable concrete components. An extension should make a new capability available to the model without forcing business authors to depend on its internal implementation.

78.2 Component and Runtime Extension Points

Good extension points are named by the capability they provide. Visual components should be selected through functional intent and registration. Runtime adapters should expose stable contracts. Implementation-specific classes may appear as bindings where the current metadata grammar requires them, but they are not business concepts and should not become the vocabulary used to describe the domain. Generated host-language bindings provide a separate programming projection of published runtime capabilities; they should not be mirrored back into the ontology or declarative vocabulary merely because an operationalization uses that language.

78.3 Common Pitfalls

Avoid four recurring mistakes:

- putting business workflow inside an opaque extension instead of a declared process or action;
- exposing an implementation helper as a public capability;
- coupling a conceptual model to one UI, transport or storage implementation when a registered boundary exists;
- treating a generated or external result as valid before it has crossed the corresponding parser, grammar or runtime validation gate.

78.4 Safe Extension Rule

A good extension adds capability without breaking the conceptual contract. The model remains understandable, the runtime can still inspect it, and the implementation can be replaced without redefining the business concept.

78.5 Summary

Extend capabilities, not accidental implementation details. Keep the conceptual surface stable, make implementation bindings explicit, and preserve validation and publication boundaries.

Chapter 79

Governing Model Evolution

Model evolution must be governed because logiCells models are executable contracts.

79.1 Change Categories

A change may add a concept, modify a role, change a facet projection, add an action, publish a service, add an event, or alter a process state. Each category has different compatibility consequences.

79.2 Compatibility

Changing a published action signature is a contract change. Changing a role path may affect persistence and views. Changing an event name may break listeners. These changes should be reviewed and versioned.

79.3 Documentation

The guide itself should follow model governance: examples stay paired, public content avoids client-specific names, and platform concepts remain clearly separated from application examples.

79.4 Summary

Governance keeps conceptual applications evolvable without losing trust in their executable models.

Part XI

Conclusion and Reference

Chapter 80

Runtime Features and Integration Checklist

This chapter is a reader-oriented compatibility checklist. It helps developers choose integration surfaces and identify configuration that must be reviewed for their own deployment. Exact commands and optional modules vary with the installed release; consult its reference documentation.

80.1 Core Developer Workflows

Area	What to check in your application
MODEL and application assembly	Validate YAML declarations, imported packages and the assembled application with the compiler you deploy.
H-Logic and hypergraph queries	Choose the query mode and contextual scope explicitly; test representative matches and rejected constraints.
Runtime objects and actions	Check identity, ownership, access control and the transaction boundaries used by each effectful operation.
Python and C# bindings	Use bindings generated for your runtime version; follow handle ownership, disposal and callback lifetime requirements.
Windows, Linux, macOS and containers	Confirm the runtime package, native dependencies and tooling required for the actual host and architecture.
MCP and OpenAPI publication	Validate generated descriptors and configure inbound authentication and permissions separately from the ability to publish an endpoint.
Agent sessions and knowledge queries	Use bounded goals, explicit context, inspected pending requests and confirmation for sensitive effects.
Compiled metadata and incremental build	Distinguish the native metadata store, semantic graph, compiled image cache and source-signature cache; use full builds when the tool reports an unsupported change.
Rich-client and distributed synchronization	Specify the synchronization provider, merge/conflict rules, idempotence and offline-retry behavior; do not infer automatic conflict-free merging.

80.2 Operational Preconditions

- Match the runtime, CLI, binding, generated descriptors and model schema to the same supported release.
- Verify the integration on each actual platform and provider combination.

- For published operations that change state, exercise allowed, denied, expired and repeated requests in a staging environment. Verify authorization and confirmation on the *inbound* execution path itself.
- Do not assume a distributed commit jointly covers SQL resources and the semantic hypergraph; define compensation and recovery where multiple resources participate.
- Evaluate performance with the application's dataset, compilation profile and hardware rather than reusing measurements from another environment.

80.3 Optional and Evolving Capabilities

Some workflows, including embedded-YAML source debugging, native extension loading and advanced knowledge-sharing operations, depend on an adapter, host platform or optional component. Use the feature list and prerequisites of the selected distribution before including them in an installation guide or a customer commitment. A development roadmap is not a runtime contract.

80.4 Maintaining Consistent Documentation

The Developer Guide and companion books share the same compatibility principles. Describe what the user can configure and verify, avoid implying platform-wide certification from one subsystem, and keep release-specific engineering evidence in internal validation records rather than in a public tutorial.

Chapter 81

What Conceptual Programming Changes

Conceptual programming changes the center of gravity of application development. It does not remove code, storage, protocols or deployment engineering; it changes which artifact owns the business meaning that those mechanisms execute.

81.1 From Code to Executable Meaning

Instead of starting from controllers, tables and routes, logiCells starts from concepts, roles, relations, actions, events and processes. Code still exists, but it implements declared meaning and runtime contracts.

81.2 From Screens to Projections

A screen is not the application. It is a projection of a concept in an interaction context. The logiCells View architecture is one optional way to implement that projection; CLI, API/service and MVP/custom clients remain valid alternatives.

81.3 From APIs to Capabilities

A service endpoint should expose a published model capability rather than become a second location where business semantics are invented. OpenAPI, RPC, ABI, MCP or another protocol can change the access boundary without changing the action's conceptual meaning.

81.4 From AI Prompts to Governed Proposals

AI becomes safer when it operates on structured concepts and published actions rather than unbounded text or raw tables. In the current framework, an H-Logic candidate can carry origin/provenance and pass through a validation lifecycle in which parsing and semantic validation remain separate from activation. A generated proposal therefore does not become knowledge merely because it parses.

81.5 From "Supported" to Evidence-Scoped Status

A mature platform description must distinguish availability, verification and architectural intent. The shared capability matrix in Chapter 80, "Runtime Features and Integration Checklist"

records those boundaries. Runtime availability on an operating system, for example, is not the same claim as validation of every native extension or ABI loader on that platform.

81.6 What Remains Conventional Engineering

Conceptual programming does not eliminate:

- performance measurement and capacity planning;
- lifecycle, ownership and deterministic shutdown;
- database and network operations;
- security configuration and deployment hardening;
- platform-specific native dependencies;
- testing of domain behavior beyond declarative syntax validation.

The benefit is that these concerns surround an explicit conceptual contract instead of silently becoming the contract themselves.

81.7 Summary

The central change is simple: meaning stays explicit and executable while interfaces, bindings and deployment choices remain replaceable projections around it.

Chapter 82

logiCells Project Layout

82.1 Scope

- applications or configurations;
- packages;
- models;
- views;
- services;
- resources;
- database templates;
- logs, cache, synchronization and snapshots;

82.2 Recommended Public Project Shape

A logiCells project should make the separation between model, view, service and deployment artifacts visible. The exact directory names may vary, but the structure should help a reader understand which files define concepts, which files define projections, which files define integration services and which files assemble the runtime application.

```
meta/models/             declarative concepts, relations, actions and processes
meta/views/contracts/    channel-independent interaction contracts
meta/views/models/       view-owned MODEL redefinitions and overlays
services/                service groups, external channels and publication settings
packages/                package descriptors used by the runtime loader
resources/               icons, translations, styles and visual assets
deploy/                  generated release trees, update descriptors and installers
```

Listing 82.1: Typical project responsibilities

The important point is not the folder name itself. The important point is that executable concepts remain independent from visual appearance and deployment details. This makes it possible to reuse a model in a desktop application, a web view, an agent process or a service publication context.

82.3 What Belongs in the Model Layer

The model layer contains the stable conceptual structure: class identifiers, roles, facets, actions, events, collections and constraints. It should not depend on a concrete visual toolkit. A model may reference a view name, but it should not encode the look of the view as business semantics.

82.4 What Belongs in the View Layer

The VIEW layer describes interaction contracts rather than one concrete screen technology. A contract can define cursors, datasets, functional components, navigation/events, component bindings and MODEL-behind references. The participant may be human or agentic. Native and web renderers can materialize controls, while an agent runtime can use the same contract to present information, ask for missing values and invoke declared capabilities without rendering a form.

The canonical generated tree has `contracts` and `models` directly below `meta/views`; historical view-folder mirrors should not be reproduced alongside them. Local manifests use `_.manifest.yaml`.

Chapter 83

Conceptual Modeling Quick Reference

83.1 Scope

- concept;
- role;
- facet;
- constraint;
- collection;
- action;
- event;
- view;
- process;
- published capability;

83.2 Core Modeling Checklist

A model should first name the concept, then declare the roles that give the concept operational meaning. After that, the model can add facets, actions, events, collections and views.

```
1. Give the concept a stable classId.
2. Declare its conceptual roles in a hypergraph facet.
3. Declare persistence projections with rolePath when needed.
4. Add actions only when they express meaningful capabilities.
5. Mark an action published only when it forms a public contract.
6. Use events to notify change, not to hide business state.
7. Use process models when several objects must be coordinated.
8. Use views as projections, not as the owner of business logic.
```

Listing 83.1: Conceptual model checklist

83.3 New Declarative Module Checklist

When starting a new module rather than editing an already-loaded model, verify the complete assembly path as well as the model itself.

```
1. Choose stable, project-owned classId values.
2. Create the package descriptor.
3. Point schemaDirectories/schemaPath to the model root.
4. Create \texttt{\_.manifest.yaml} and declare every loadable model.
5. Create the declarative models.
6. For semantic entities, declare the Entity concept explicitly.
7. Do not add a literal Me field; entity Me is implicit semantically.
8. Declare Actions, Events and Processes as model capabilities.
9. Create a separate test package and test models.
10. Check for duplicate classId values in the assembled application.
11. Start or compile the application from its .lgc configuration.
12. Verify model compilation and semantic initialization in the log.
```

Listing 83.2: New module checklist

The most important distinction is between a source file and an assembled application model. A correct file that is not reached through the package and manifest loading chain is not part of the running application.

83.4 Namespace Guidance

The namespace `lgc#` denotes platform concepts and reusable runtime concepts. A namespace such as `lbs#` may be used for example application concepts. The distinction helps the reader see what is part of the platform and what is part of an application domain.

83.5 Model Smells

A model becomes difficult to maintain when scripts contain hidden orchestration, when views own business rules, when database fields replace conceptual roles, or when processes are represented only by button handlers. These smells are often solved by introducing a process concept, a relation concept, a published action or an explicit event.

Chapter 84

H-Logic Quick Reference

84.1 Definitions and Instances

```
define .:#Person(me)[name::#string];
#alice:#Person;
_:#Person;
```

Listing 84.1: Core forms

84.2 Roles, Attributes and Order

```
define .:#worksFor(person:#Person, company:#Company, context:#Context);
define .:#Order(me)[amount::#float, status:#OrderStatus];
define .:#Evidence(item:#Item, context^2:#Context);
```

Listing 84.2: Structural and descriptive positions

84.3 Interfaces and Named Terms

```
doc42::#Auditable;
projectBeta := #projectBeta;
exists target=&projectBeta on .:#VisibleIn(label42, target);
```

Listing 84.3: Attachment and named-term binding

84.4 Constraints

```
define .:#Required(x:#A not null);
define .:#Defaulted(x:#A default &a);
define .:#Fixed(x:#A=&a);
```

Listing 84.4: Local term constraints

84.5 Logical Relations

```
#A ->_[weight=1, context=#c] #B;
#A <->_[context=#shared] #B;
#Diagnosis :-_[confidence=0.8] #Pattern;
#Candidate !-_[reason=#invalid] #Bottom;
```

Listing 84.5: Logical operators and complements

84.6 Quantifiers and Queries

```
any x:#A, exists y:#B on .:#Pair(x, y);
every y:#B on .:#R(y);
ask .:#A(x) and .:#R(x, y) return x;
? .:#Person(Tom)[name='Tom Hanks'] return Tom;
```

Listing 84.6: Quantified and query forms

84.7 Query Modes and Scope

```
ask homomorphic subtypes instances .:#Document(d)
  return d;

ask unifying (.:#Pair(x, #b), .:#Pair(#a, y))
  return (x, y);

ask resolving .:#reachable(#a, target)
  return target;
```

Listing 84.7: Reasoning mode and scope options

Mode or option	Meaning
`homomorphic`	Persisted structural pattern matching; default mode.
`unifying`	Canonical unification of two or more compatible operands.
`resolving`	Goal resolution through facts and Horn-style rules.
`instances [only]`	Include or restrict results to instances.
`types [only]`	Include or restrict results to types.
`subtypes [only]`	Traverse the requested type and its descendants.
`supertypes [only]`	Traverse the requested type and its ancestors.
`masked`	Apply active masking or visibility restrictions.

Result shape. Zero answers produce `nil`; one answer is returned directly; several answers form a linked conceptual list. A tuple or list written in `return` is not wrapped again.

Inherited values. Descendant instances may be returned through mapped views. The real instance identity is preserved while slots are exposed in the requested ancestor order. The normal polyadic map, not a reduced permutation, is the semantic source of truth.

84.8 Composite Definitions

```
define .:#ContextualRule(rule, context) as
  .:#Rule(rule)
  and .:#Context(context)
  and .:#validIn(rule, context);
```

Listing 84.8: Logical composition

84.9 Forms to Avoid

- do not use `&x` as a role name;
- do not combine quantified typing and equality as `y:#A=&a`;
- do not write query variables as `?x`;
- do not encode context as an unsupported function call;
- do not use a structurally undefined type in an application or interface attachment;
- do not expose unconstrained public roles in a projected definition.
- do not interpret a wildcard as an exact `nil` constraint;
- do not assume a subtype instance has the same visible slot order as its parent;
- do not add an artificial list wrapper around a singleton query result.

Chapter 85

H-Logic Tested Patterns

The examples in this appendix are aligned with H-Logic grammar version 3.4 and the current parser/evaluation test categories. They are intentionally small so that each pattern demonstrates one stable language feature.

85.1 Primitive Values and Collections

```
#an_apple;
1;
'abc';
1.75;
true;
();
(x,);
(x, y);
[a, b];
{#a, #b};
```

Listing 85.1: Primitive expressions and collections

85.2 Definitions, Roles and Attributes

```
define .:#P0;
define .:#Person(me)[name::#string];
define .:#worksFor(person:#Person, company:#Company, context:#Context);
define .:#Transfer(me)[sender:#Person, receiver:#Person, amount::#float];
```

Listing 85.2: Type definitions

85.3 Named and Anonymous Instances

```
define .:#Person(me)[name::#string];
#alice:#Person;
_:#Person;
.:#Person(Tom)[name='Tom Hanks'];
```

Listing 85.3: Instance patterns

85.4 Interface Attachment

```
define .:#Customer(me)[name::#string];
#alice::#Customer;
_:#Customer(#alice);
```

Listing 85.4: Boxing and attachment

85.5 Quantifiers

```
exists y:#B on .:#R(y);
every y:#B on .:#R(y);
any x:#A, exists y on .:#Pair(x, y);
any x:#A, every y:#B on .:#Pair(x, y);
any x:#A, exists y:#B, exists z:#C on .:#R(x, y, z);
```

Listing 85.5: Quantified patterns

85.6 Logical Relations and Complements

```
.:#a(x) -> .:#b(x);
#A ->_[weight=1, context=#c] #B;
#Diagnosis :-_[weight=0.8, context=#clinical] #SymptomPattern;
#Candidate !-_[context=#validation] #Bottom;
```

Listing 85.6: Logical relation patterns

85.7 Composite and Projected Definitions

```
define .:#f(x, y) as .:#a(x) and .:#c(x, y);
define .:#fExists(x, y) as
  .:#u(x) and exists y:#B and .:#v(x, y);

define .:#PersistedU(me, x);
define .:#PersistedV(me_, x, y);
define .:#ProjectedF(x, y) as
  .:#PersistedU(me, x)
  and .:#PersistedV(me_, x, y);
```

Listing 85.7: Composite definition patterns

85.8 Commands and Queries

```
projectBeta:#Context;
tell projectBeta:#Context;
do projectBeta:#Context;
ask .:#A(x) and .:#R(x, y) return x;
```

Listing 85.8: Command patterns

85.9 Negative Test Categories

The test suite also rejects clarified invalid forms. Public documentation should preserve these boundaries:

- query variables prefixed as `?x`;

- explicit named terms used as role names;
- unsupported context function calls;
- structurally applied types that have not been defined;
- projected public roles absent from the composite body.

85.10 Documentation Rule

Before adding an H-Logic example to the guide, verify it against the current grammar and, when possible, add or identify a corresponding parser or evaluation test. This turns H-Logic documentation into executable specification rather than illustrative pseudo-syntax.

Chapter 86

Declarative Action Patterns

86.1 Scope

- object action;
- class action;
- published action;
- service-exposed action;
- event handler action;
- process transition action;

86.2 Object Actions

An object action belongs to an instance. It is used when the capability depends on the state of one conceptual object. Typical examples are `Start`, `Validate`, `Finalize`, `Retry` or `Open`.

```
actions:
- action:
    name: Validate
    type: ObjectItem
    published: true
    params:
    - Result:
        dataType: Boolean
        paramType: ptReturn
```

Listing 86.1: Object action pattern

86.3 Class Actions

A class action belongs to the concept itself. It is useful for factories, searches, publication endpoints or domain operations that do not depend on an already selected object.

86.4 Published Actions

A published action is a contract. It can be used by a view, an agent, another process or a webservice publication layer. The declaration of parameters is therefore part of the public interface of the conceptual model.

Chapter 87

Conceptual Event Patterns

87.1 Scope

- business event;
- visual event;
- process event;
- message event;
- local event;
- remote event;

87.2 Event as Observable Change

An event should announce that something meaningful happened. It should not be a hidden substitute for business state. The event name, source, parameters and bound actions form a declarative routing contract.

```
events:
  - lbs.Process.Completed:
      actions:
        - action:
            path: ./Action[@name='OnCompleted']
```

Listing 87.1: Event-to-action binding pattern

87.3 Local, Visual and Remote Events

A visual event coordinates a view or component. A business event coordinates conceptual objects. A remote event crosses a process, agent or messaging boundary. Keeping these categories separate helps avoid coupling the interface to the business process.

87.4 Process Events

A process model should emit events such as `Started`, `Validated`, `Completed`, `Rejected` or `Failed`. These events make orchestration observable and allow controllers, agents and services to react without embedding the workflow in one procedural script.

Chapter 88

Workflow Patterns

88.1 Scope

- synchronous process;
- asynchronous process;
- human validation process;
- message-driven process;
- service-backed process;

88.2 When to Create a Process Model

Create a process model when a sequence crosses more than one business object, when state must be observed, when failure must be handled, or when a user, service or agent must resume work later.

88.3 Synchronous Workflow

A synchronous workflow is a sequence of actions executed by one process action. It is appropriate when the process is short, deterministic and can complete in one execution context.

88.4 Asynchronous Workflow

An asynchronous workflow emits events or messages and waits for participants to respond. It is appropriate when coordination crosses user interaction, service calls, agents, messaging queues or long-running activities.

88.5 Anti-Pattern

Do not hide a cross-object process in a view button. The button may start the process, but the process model should own the orchestration.

Chapter 89

AI Assistant Patterns

89.1 Scope

- model explainer;
- process summarizer;
- service caller;
- test generator;
- trace interpreter;

89.2 Assistant over Concepts

An assistant should operate on concepts, actions, rules and views rather than on unstructured screens. The model gives the assistant stable names, typed roles and executable capabilities.

89.3 Read, Explain, Act

A useful assistant pattern has three layers. First, it reads conceptual state through queries and views. Second, it explains the state using model names. Third, it acts only through declared actions, preferably published actions with explicit parameters.

89.4 Guardrails

The assistant should not bypass the conceptual runtime. Authorization, validation, process state and traceability remain part of the model. This is what makes AI integration compatible with governance.

Chapter 90

Runtime Recipes

90.1 Scope

- create object item;
- find class item;
- generate provider;
- open buffer;
- post data;
- build event;
- send event;

90.2 Create an Object

Use the runtime container to create an object from a declared concept. Set declared roles, then persist the object through the normal persistence mechanism.

```
Item := ObjectsContainer.CreateObjectItem('lbs#work_item', 1, '');
Item.Name := 'Example';
Item.PostData();
```

Listing 90.1: Runtime creation recipe

90.3 Find a Concept

Resolve a class item by its class identifier when the script must work through the conceptual model rather than a hard-coded implementation class.

90.4 Execute a Process

Create the process object, attach the participating business objects, then call the published starting action. The process is responsible for subsequent events, state changes and trace entries.

Chapter 91

functionalP Language Essentials

This appendix summarizes the language features behind the practical subset introduced in Chapter 37, “functionalP for logiCells Beginners”. It is intentionally more language-oriented. New logiCells developers should first learn the metadata patterns, then return here when they need to understand a script that uses lists, functions, lambdas or advanced control flow.

91.1 Purpose

`functionalP`, often abbreviated as FPL in runtime material, is a compact runtime language used to express computations, action bodies, process fragments and graph-oriented operations. It is not intended to replace the declarative model. It provides a programmable layer that can be stored in metadata, executed dynamically and connected to containers, actions, services and conceptual structures.

This introduction follows the FunctionalP grammar supported by the target runtime. The examples focus on language constructs a developer can use when writing and debugging actions; check the syntax against the compiler distributed with your project.

91.2 Program Structure

A program may include an optional library/import header followed by a sequence of terms.

```
1 library Runtime.Demo;
2 import Runtime.Core;
3
4 Result := 1 + 2;
```

Listing 91.1: Library and import header

Most logiCells metadata scripts do not need a header:

```
1 5 + 10;
```

Listing 91.2: Minimal expression

91.3 Sequences

A script is usually written as a sequence of expressions separated by semicolons.

```
1 t1 := 5;
2 t2 := 8;
3 t1 - t2;
```

Listing 91.3: Sequence of assignments and final expression

Intermediate expressions may update the runtime environment. The final expression may become the script value. When a stable return value matters, assign `Result` explicitly.

91.4 Primitive Values

The language supports common primitive constants:

```
'Hello world!';
42;
42.65;
true;
false;
nil;
```

Listing 91.4: Primitive values

Runtime tests may format booleans as `T` and `F`. Application scripts should use boolean values, not formatted strings.

91.5 Assignments and Variables

The assignment operator is `:=`.

```
a := 'Bonjour';
a;
```

Listing 91.5: Assignment and retrieval

Assignments are commonly used to name datasets, object items, events, classes and intermediate values.

91.6 Arithmetic, Comparisons and Boolean Logic

The grammar supports common arithmetic operators, comparison operators and boolean composition.

```
5 + 10 * 2 + 6;
5 <> 10;
5 <= 5;
true and false;
false or true and false;
```

Listing 91.6: Arithmetic and boolean expressions

`functionalP` is parsed as an expression language. It is not a textual macro language.

91.7 Lists and Ranges

Lists are written with comma-separated expressions. The empty list is `()`.

```
(1, 2, 3);
();
(1, 2, (5, 4, 6));
1 .. 5;
```

Listing 91.7: Lists and ranges

The language also exposes list-oriented operators such as `head`, `tail`, `last`, `length`, `reverse`, `drop` and `take`.

```
tail (1, 2, 3);
length (1, 2, 3, 4);
reverse (1, 2, 3);
```

Listing 91.8: List-oriented operators

91.8 Control Flow

The language supports conditional and loop constructs.

```
a := 0;
if (a = 0) then
begin
  b := 1;
end
else
begin
  b := 2;
end;
b;
```

Listing 91.9: Conditional expression

```
res := 0;
for value in (1, 2, 3, 4) do
begin
  res := res + value;
end;
res;
```

Listing 91.10: For loop over a list

Use loops sparingly in metadata scripts. If a script becomes complex, consider a declared action, process or native implementation.

91.9 Functions

Functions are declared with `function`, a name, a parameter list and a body.

```
function double(a)
begin
  a * 2;
end;

double(4);
```

Listing 91.11: Simple function

A function may be used with functional operators:

```
function double(a)
begin
  a * 2;
end;

map double (1 .. 5);
```

Listing 91.12: Mapping a function over a range

91.10 Partial Application and Lambdas

Some tests exercise partial application and lambda expressions. Treat these as advanced features in application metadata.

```
(lambda(x) x * x) 4;
```

Listing 91.13: Lambda expression

Prefer explicit named actions for business behavior. Lambdas are useful for local transformations, not for hiding durable domain rules.

91.11 Case Expressions

Case expressions use pattern-like clauses introduced by `:-`. Clauses may be separated by `|`.

```
function normalize(a)
begin
  case a of
    () :- ()
  | _  :- a
  end;
end;

map normalize (((1,2)), (), ((3,2)));
```

Listing 91.14: Case expression with empty-list fallback

The underscore is a wildcard-like pattern.

91.12 Dot Expressions and Runtime Objects

`functionalP` can call object methods and properties exposed by the runtime environment.

```
Logger.WriteLog('functionalP script executed', 1);
```

Listing 91.15: Calling a runtime method

A script may also interact with injected objects:

```
aInstance.Name := 'RuntimeObjectName';
aInstance.Name;
```

Listing 91.16: Reading and writing a runtime object property

In logiCells metadata, dot expressions most often access `Self`, `Sender`, `ObjectsContainer`, `PageControler` and `DataSetsManager`.

91.13 Hypergraph Literals and Process Constructs

The grammar includes graph-oriented and process-oriented constructs. They are important for the runtime but should be treated as advanced material in an onboarding guide.

Use tested H-Logic, model declarations or runtime APIs for production code unless the literal form is explicitly documented for the module you are using.

91.14 Common Misreadings

Misreading	Correction
`functionalP` is the model language	It is an executable scripting layer used inside YAML metadata.
It has the same rules as Object Pascal	It borrows some surface syntax, but it is a runtime expression language with its own grammar and evaluation model.
Every runtime test is a production recipe	Tests demonstrate supported features and edge cases. Refactor examples before using them in onboarding or production documentation.
Formatted boolean output is part of the business contract	Use typed boolean values. Do not depend on test formatting such as `T` or `F`.
The final expression is always obvious	Assign `Result` explicitly when the action contract expects a return value.

91.15 Recommended Learning Order

1. simple expressions;
2. assignments and `Result`;
3. `Self`, `Sender` and injected runtime objects;
4. dataset and page-controler scripts;
5. functions and lists;
6. lambdas and case expressions;
7. process constructs only after the expression model is clear.

Chapter 92

Glossary of Conceptual Programming

92.1 Scope

- Concept: named unit of meaning;
- Facet: operational projection of a concept;
- Role: typed position inside a concept or relation;
- Action: executable intention;
- Event: observable change;
- Process: coordination object;
- View: contextual projection;
- Functional component: declared interaction function;
- Published action: externally accessible capability;

92.2 Selected Terms

Concept. A declared unit of meaning that can carry roles, constraints, facets, actions, events and views.

Conceptual memory. The graph or hypergraph layer where conceptual structures can be represented, queried and evaluated.

Facet. A projection of a concept for a specific concern, such as hypergraph structure or database persistence.

Published action. A declared action marked as externally accessible as a capability. It may be used by views, processes, agents or webservices.

Functional component. A declarative view component that names a user-interface function independently of a concrete implementation.

Process model. A model whose purpose is to coordinate several objects, actions, events or participants.

Messaging platform. The reusable platform mechanism for asynchronous message-based coordination.

Chapter 93

Troubleshooting Guide

93.1 Scope

- check class identifiers;
- check role paths;
- check provider filters;
- check action parameters;
- check event listeners;
- check view paths;
- check service publication;
- check YAML parity;

93.2 A View Does Not Show the Expected Data

Check the cursor first, then the dataset, then the collection path. Verify that the role paths used by the collection correspond to the roles declared in the model.

93.3 An Action Cannot Be Called Externally

Check that the action is marked as published, that its parameters have declared types and that the publication layer is allowed to expose the owning concept.

93.4 A Process Is Hard to Debug

Look for hidden orchestration in view scripts or object actions. Move cross-object coordination into a process model, then emit explicit events for major state changes.

93.5 A Component Is Too Platform-Specific

Replace the concrete component assumption with a functional component declaration. Register the concrete implementation separately for each target platform.