

Architecture Harness

SimpleModeling

Software architecture is the fundamental design that defines a system’s mechanisms, their roles, and how they work together to deliver the required functionality and quality. SimpleModeling provides a reference architecture in which software is described at an abstraction level close to domain knowledge, and those models form the basis for development and execution. This article places the technologies covered in this ten-part series within that overall design. It explains how their combination aims to improve development speed and reduce AI costs while maintaining the required quality. By arranging multiple safeguards from development through execution, this design also functions as an architecture harness for AI development.

1 Architecture and Development

Architecture defines the technologies to use, their roles, and their relationships in order to deliver the required functionality and quality. SimpleModeling provides a reference architecture as a common design for this purpose.

The modeling method can be adopted on its own, and Cozy and Textus can also be used independently. Model checks, Cozy’s generation rules, Scala’s type checks, and Textus’s execution rules each act as a harness that constrains AI-generated changes and their execution. The corresponding articles explain how each technology applies these safeguards.

The reference architecture shows how these technologies can be combined to optimize development as a whole. Models and generation rules reduce custom implementation, components limit the scope of changes, and type checks reduce the problems that reach review. A shared runtime then applies constraints during execution. This division of responsibilities aims to maintain the required quality, improve development speed, and reduce AI costs. It also limits departures from requirements and rules, providing an environment in which AI is more likely to arrive at a correct solution. The effects of this combination are the subject of the architecture harness discussed here.

1.1 Time and Cost

AI-assisted development incurs time and cost for checking specifications, reviewing changes, and making corrections, as well as generating code. An AI review requires inference to examine specifications, surrounding code, and dependencies. Reviewing a correction again also takes inference time and incurs usage costs.

Generation rules and type checks detect mechanically decidable problems before review. Component boundaries limit the scope of changes and verification. Combining these mechanisms can

reduce the material to review and the number of correction cycles, lowering development time and AI usage costs while checking conformity with requirements.

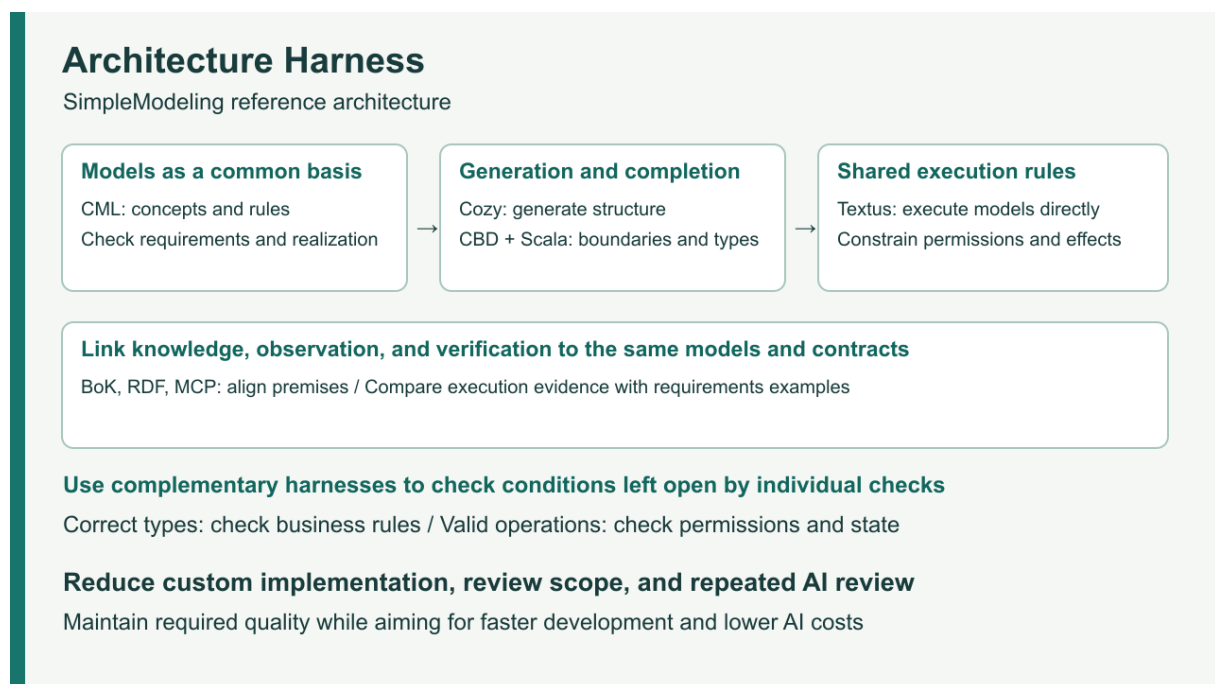
1.2 Reference Architecture as a Harness

SimpleModeling starts with model-driven development. It combines a modeling language, components, a programming language, and mechanisms for generation, checking, and execution so that software can be built and run while preserving the meaning of its models. This common design is the reference architecture.

In AI development, this design defines where AI judgment is needed and where generation rules, types, and execution rules apply. Viewed in an architectural context, skills that describe work procedures are one part of the harness. The scope includes the entire development and execution environment that helps AI produce appropriate results.

2 The Reference Architecture

SimpleModeling's reference architecture uses models of domain concepts, relationships, and rules as a common basis for generation, supplementary implementation, execution, and verification. Cozy derives structure from models written in CML; AI supplies missing behavior within the resulting contracts; and Textus runs executable models. Component-based development (CBD) assigns responsibility for models and implementations to components, while knowledge and execution evidence are linked to the same models. This arrangement reduces repeated interpretation of rules at each stage and narrows what must be examined when a change or inconsistency occurs.



2.1 Shared Models and Generation

CML describes models, Cozy derives structure from them, and Textus handles execution. Textus directly executes rules expressed in models, while Cozy generates the surrounding connections. This reduces the work of having AI rewrite both the rules and their connections separately. Applying model-derived types and operation contracts to supplementary code also limits implementations based on assumptions that differ from the model.

CBD defines components with public operations and conditions of use. A three-layer structure organizes their responsibilities: presentation handles screens and dialogue; application handles the composition of processing along use cases; and domain handles the subject's concepts, states, and rules. Domain models center on static models, while application models center on workflows starting from use cases. Combining these two divisions allows new screens to use existing business operations and reduces the need to reimplement and recheck rules for each screen.

2.2 Execution and Feedback

Application workflows call public domain operations and progress according to state machines and business rules. CLI, REST, and MCP serve as entry points to the same processing. Consolidating business logic, authorization, and access to external resources into shared paths reduces the opportunity for entry points to apply different decisions. Specifications and verification results for the common processing can be reused, while input conversion and the passing of authentication information are checked for each entry point.

Knowledge used in development carries links to terms, models, and specifications, together with its sources and versions. Execution results are expressed in the vocabulary of those same models. Combined with requirements verification, these links help identify whether an unexpected result calls for examining the knowledge used, the rules described, or the processing executed. They reduce the need to reread models, logs, and documents separately and infer their relationships, supporting corrections based on evidence.

2.3 The Ten-Part Series

Episode 1 discusses the possibilities of AI development, the conditions required for business applications, and the cost of reaching an acceptable result through verification, correction, and approval. Episode 5 examines how the technologies in the series are combined to meet those conditions.

The model harness in Episode 2 builds on CML, model-driven development, model up/down, and the correspondence between requirements and their realization. Episode 3 covers static models, including entities, value objects, aggregates, associations, powertypes, traits, resources, and task classifications and constraints. Episode 4 covers dynamic models such as workflows, state machines, and use cases. Comparing operations, states, and procedures helps detect in-

consistencies and correct them before implementation.

Episode 6 constrains supplementary implementations through Scala’s types, algebraic data types, immutable data, purely functional programming, higher-order functions, monadic computation structures, internal DSLs, Free Monads, and interpreters. Episode 7 covers execution of models in Textus, together with events, authorization, external effects, concurrency, retries, and compensation. Operations permitted by the language are connected to runtime paths. Types and similar mechanisms detect problems that can be decided before execution, while the runtime checks conditions that depend on the user or current state. Together, they reduce the authorization and failure-handling logic that must be written separately in each piece of AI-supplied code.

The knowledge harness in Episode 8 connects literate models and object models through a BoK glossary, manages sources, versions, and applicability with RDF, and supplies knowledge to AI through MCP. Aligning knowledge with the target models and component contracts aims to reduce implementations based on obsolete specifications or different conditions of use that later require substantial correction.

Episode 9’s observation models bring runtime evidence back to the vocabulary and abstraction level of models. Episode 10’s verification harness covers executable specifications, tests, AI review, and human judgment and approval. Relating expected results to actual operations and transitions helps identify both that a result differs and where a correction belongs. Checks and diagnostics from the individual technologies support corrections appropriate to the cause. Their detailed mechanisms are left to their respective articles; this article explains how they interact.

3 Models as an Execution Basis

Models describe domain concepts, relationships, and rules at an abstraction level close to business knowledge. Using that structure for checking and execution reduces the translation of the same rules into individual pieces of code and the work of comparing that code with the model.

3.1 DDD Models and Direct Execution

Entities, values, aggregates, and operations from domain-driven design (DDD) are modeled as building blocks whose business meaning is clear. Dynamic processing is described through workflow sequences and branches, and through state-machine states, events, and transition conditions. Textus directly runs executable models, so execution follows these definitions.

Less rewriting of rules into individual flags and conditional branches means less supplementary code for AI to produce and less comparison between models and code. Models can be treated as programs in the AI era because their high-level descriptions serve as the basis for checking and execution.

3.2 Requirements and Rules

Use cases are requirements models that describe user goals and concrete examples. They are mapped to the workflows, state machines, and domain operations that realize them, rather than being executed directly. Model review through `textus-cbd-support` checks the realization links and structural consistency, while executable specifications check the execution results of the same examples. Together, these mechanisms help find missing realization at the model stage and verify whether execution produces the intended results.

Business rules can be collected in domain-side models and evaluated by a rule engine. Defining rule ownership through CBD and sharing the evaluation mechanism reduces the opportunity to write different decision logic for each screen or operation. CNCF has a rule-engine foundation, but mapping domain rules to the runtime as first-class model elements remains a design issue.

3.3 Generation and AI Completion

Cozy generates interfaces and connections that can be mechanically derived from models. Deriving structure through the same rules reduces the work of reviewing and reconciling different structures that AI might otherwise choose for each feature.

AI supplements the models with programming-language code for calculations and external-service connections that the models cannot fully express. Developers work primarily with domain models, while generated types, operation contracts, and component usage rules apply to the supplementary code. When a model changes, its structure can be regenerated, and supplementary code that no longer matches the types or contracts can be detected and corrected. Combining generation and checking reduces the scope for AI to guess what changed and rewrite the entire implementation.

4 CBD and Scala

When models are executed directly, code supplies processing that the models cannot express, and checks examine that code and the connections between components. CBD divides responsibilities, while Scala's types and computation disciplines apply within those boundaries. This allows inter-component inconsistencies and local implementation errors to be examined separately.

4.1 Public Contracts

Each component defines capabilities, public services and operations, their inputs and outputs, and conditions of use. External consumers use the public API, and replaceable implementations follow an SPI. Preventing direct dependencies on internal data and defining permitted dependencies reduces the paths through which an internal change can affect other components.

Linking public contracts to model-derived types and executable specifications allows consumer connections and expected behavior to be checked separately when AI changes a component's internals. For changes that preserve the contract, verification can focus on the responsible component and its contract, reducing repeated examination of the entire system. If the contract itself changes, dependencies identify the consumers whose verification must be expanded.

4.2 Scala and Review Cost

Scala's type checking is itself a harness for supplementary implementation. Combined with model-derived types and CBD public contracts, it can detect, before review, type-expressible errors such as passing a value with a different meaning or returning a result that violates a contract. Algebraic data types describe operation results and states, allowing compiler diagnostics to identify missing cases where supported.

Immutable data and pure functions limit where values can be changed and where side effects can occur. Within a component already bounded by CBD, processing can be understood in terms of inputs, outputs, and explicit effects. This reduces the states and execution paths that reviewers must follow.

The combination aims to improve development speed and reduce AI costs by limiting review and correction cycles caused by mechanically checkable defects. These are design effects, not measured reduction rates. The correctness of requirements and business meaning must still be checked separately against models and examples.

4.3 DSL and Runtime

An internal DSL defines the vocabulary of operations available to a program. With a Free Monad, combinations of operations are represented as values, and an interpreter performs external effects. Relating model-defined operations, Scala types, DSLs, and shared runtime functions reduces implementations that access external resources from arbitrary locations.

Routing operations permitted by the DSL through common authorization and resource-access paths separates the code that describes operations from the mechanism that permits their effects. This reduces scattered permission and limit checks in supplementary code and consolidates the changes needed when a policy changes. Episode 6 covers type and DSL constraints; Episode 7 covers runtime conditions and checks for paths that bypass them.

5 Complementary Harnesses

Each harness has a limited scope. An implementation may pass type checking yet violate business rules; an operation consistent with a model may still be disallowed by runtime permissions or current state. Model checks, type checks, contract checks, runtime constraints, and requirements-based verification are combined so that another harness checks conditions outside the scope of any one mechanism, covering gaps in constraints.

The reference architecture defines the scope of each harness and which mechanisms check the remaining conditions. Model checking examines structure and correspondence; type checking examines the consistency of values and operations; runtime checks examine user permissions and current state. Execution results are compared with examples from the requirements. Handling mechanically checkable problems early and checking the remaining conditions later combines multiple constraints with development efficiency.

The models, contracts, and implementations being checked have specified versions and conditions of applicability. When a target changes, dependencies identify the affected checks to rerun. Making the reuse conditions of earlier results explicit reduces duplicate checks while ensuring that checks required after a change are performed.

5.1 Components and Composition

Model structure and correspondence are checked before generation and type checking based on those models. Diagnostics caused by a model are addressed in the model; type violations in supplementary code are corrected in the responsible component. This avoids individually repairing many generated locations derived from an incorrect model and repeatedly asking AI to review problems that types can decide.

A CAR groups implementation, contracts, dependencies, configuration, and specifications into one distribution and inspection unit. A SAR describes a composition of components. Package validation and lint check required structures, dependencies that can be expressed as rules, and prohibited patterns. The validity of individual components is checked separately from dependency and configuration consistency in a composition, and the results become admission conditions. Identifying the affected component or composition reduces the need to investigate the entire configuration later.

5.2 Runtime Constraints

Static checks examine operation connections and types, while runtime checks examine user permissions and current state. Authorization and workflow/state-machine decisions are placed on public-operation execution paths, rejecting calls that violate permissions or transition conditions. Dividing decisions between build time and runtime reduces the amount of such logic AI must implement separately in each caller.

The design for external-resource effects combines early detection through CAR lint, admission checks at loading, and sandboxing appropriate to the execution context. Relating resource-access constraints to CBD boundaries limits the impact of incorrect changes and operations. Isolation of malicious code requires separate process or container isolation. Episode 7 covers the specific scope and implementation status.

5.3 Knowledge and Skills

The versions and applicability of business knowledge used as implementation premises also need checking. The knowledge harness aims to connect literate and object models through a BoK glossary, retain sources, versions, and scope through RDF, and align knowledge supplied through MCP with the development target. Combined with CBD contracts, it can supply AI with knowledge and usage conditions relevant to the responsible component, reducing specification searches, reinterpretation, and rework caused by incorrect premises.

Development skills state which models and contracts to consult, the permitted scope of changes, and how to respond to check results. Consistent knowledge aligns the premises for judgment, while generation, checking, and execution constraints examine the result. Mapping skill procedures to processing-system checks makes departures detectable. When AI is used within an application, it interprets ambiguous information or generates candidates, while workflows and the runtime handle routine progression and state changes. This reduces the inference needed for AI to reconstruct the entire procedure on every execution and the opportunities for deviation.

5.4 Observation and Verification

Observation models map evidence about executed operations, state transitions, and failures to domain and use-case vocabulary. Combined with the links between requirements and realization, these observations reveal where actual progression differed from expected behavior. Model up returns evidence to an abstraction level suitable for judgment, reducing repeated interpretation of the relationship between logs and code and directing corrections to models or responsible components.

Executable specifications are contracts that check inputs, outputs, premises, examples, and expectations on failure through execution. CLI, REST, and MCP share specifications for the same operations, while separate conformance tests examine AI's interpretation of intent. After locating a failure, AI review and human judgment assess semantic validity and whether exceptions are acceptable. Approved changes are then reflected in the models and knowledge.

6 An Order Processing Example

Consider a design example in which an order is confirmed after inventory reservation and payment authorization. It illustrates how requirements, models, supplementary implementation, execution, and verification correspond.

6.1 Models and Component Contracts

Use cases describe the buyer's goals and examples, including insufficient stock and payment failure. Static models define orders, line items, and amounts. A workflow defines inventory reservation, payment authorization, and order confirmation; a state machine defines order states and transition conditions. Model checks examine whether the operations needed to reach the required results are present.

The domain owns pricing and tax rules, and the application workflow uses a public operation to obtain the price. If the design connects those rules to a rule engine, evaluation is also concentrated inside that operation. Aligning the rule model, its owning component, and the operation that uses it is the design that prevents separate calculations for each screen.

6.2 Implementation and Execution

Cozy generates the connections between operations, while AI supplies details such as the payment-service connection. Types distinguish amounts, order identifiers, and processing results. CAR validation and lint check dependencies and operation usage rules. For example, compilation detects a supplementary function returning the wrong result type, while lint and similar checks examine dependencies on another component's internals where such rules have been defined. Both examine the same supplementary processing, allowing the diagnosed location to be corrected and its connections and dependencies to be checked again.

At runtime, authorization checks whether the user may confirm the order, and Textus advances processing according to the workflow and state machine. Payment failure prevents confirmation and leads to the defined failure handling or compensation for the inventory reservation. External-service calls also use shared effect paths so that permissions and execution conditions that types alone cannot determine can be applied.

6.3 Evidence and Change

Executable specifications check successful order confirmation and the result of payment failure. An observation model records the operations and transitions taken. Evidence narrows the correction target: a workflow for execution-order problems, rules or supplementary code for amount calculations, and policies for user permissions.

In this order process, an amount can have the correct type while the applied pricing or tax rule is wrong. The rule model is compared with requirements examples, and executable specifications check calculated results. Likewise, a confirmation call can satisfy types and contracts while payment remains unauthorized; state and workflow conditions then stop it. Authorization stops a call when the user lacks permission. Applying different constraints to the same processing covers inconsistencies and permission conditions that remain after a single check has passed.

7 Quality and Efficiency

SimpleModeling’s reference architecture combines CBD, Scala, generation, checking, execution infrastructure, knowledge, observation, and verification around models. Defining the role of each technology aims to improve development speed and reduce AI costs while maintaining the required quality. The placement of multiple safeguards is part of this optimization of the whole design.

7.1 Combined Benefits

Individual harnesses prevent errors within their own scope, such as models, components, implementations, or execution. The reference architecture connects these scopes through shared models and contracts. Direct execution and generation reduce supplementary code; CBD divides that code into components; types check their internals; and a shared runtime executes permitted operations. Reducing custom implementation, verification scope, and problems entering review can reduce AI inference and correction cycles at each stage.

The division of responsibilities is combined with mechanisms that cover the constraint gaps left by individual harnesses. Generation, checking, and execution infrastructure handle mechanically checkable conditions, while review examines meaning and conformity with requirements. This broadens coverage while reducing repeated AI inference and correction. It also helps preserve structure, limit the impact of incorrect changes, and support reimplementations from public specifications. Effects can be examined through elapsed time and AI usage through verification, correction, repeated review, and approval.

Diagnostics and execution evidence also help AI identify where a correction belongs. Model errors return to the model; type mismatches to supplementary code; and permission inconsistencies to call conditions or policies. Changes to requirements or contracts themselves require judgment and approval separate from implementation corrections. Specifying correction scope and checks corresponding to each diagnostic limits AI changes spreading to unrelated areas.

7.2 Further Articles

The roles of individual technologies as harnesses are explained in Episodes 2 – 4 on models, Episode 6 on Scala, Episode 7 on the runtime, Episode 8 on knowledge, Episode 9 on observation, and Episode 10 on verification. The reference architecture defines how they share models and contracts to reduce duplicate implementation, inference, and review. Their constraints and diagnostics limit AI’s scope of change and provide an environment for correcting results toward conformity with requirements.

CNCF combines execution of DDD-level models with reactive processing, high availability, security, and AI use to address the quality requirements of business systems. The specific configurations and implementation stages will be discussed in the subsequent CNCF series.