

Architecture Harness

The SimpleModeling reference architecture

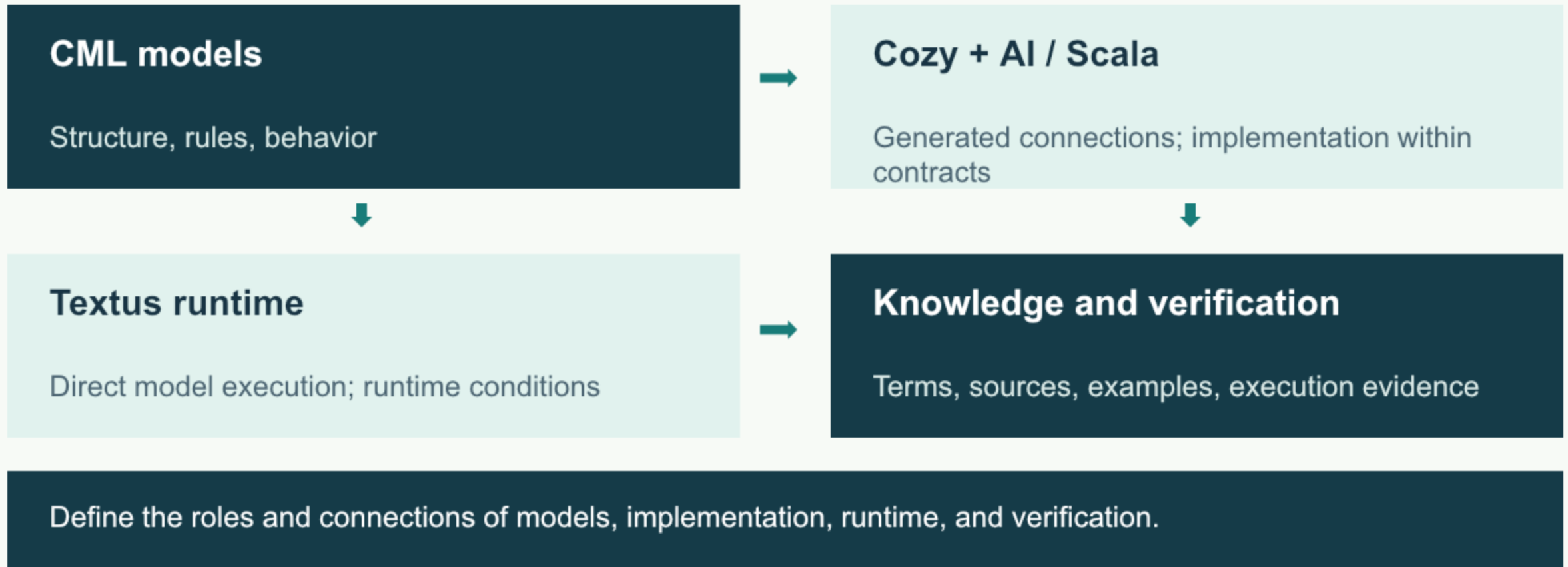
Combined mechanisms

Quality, development speed, and AI cost

Complementary harnesses

Cover conditions left open by individual checks

The reference architecture as a whole



Independent technologies in a common design

Independent use

Modeling methods, Cozy, and Textus



Each mechanism is a harness

Model checks, generation rules, types, execution



Optimize the whole process

Quality, development speed, and AI cost

Combine independently usable technologies to optimize development as a whole.

Models guide generation and execution

CML and Textus

Describe models and execute them directly



Cozy

Generate interfaces and connections



AI and Scala

Supply missing behavior within contracts

Shared models and contracts constrain generation, implementation, and execution.

Component boundaries and three layers

Presentation

Screens, dialogue, and input conversion



Application

Workflows starting from use cases



Domain

Models of concepts, structure, and rules

Layer responsibilities and public contracts define the scope of AI changes.

High-level models for checking and execution

DDD-level building blocks

Entities, values, aggregates, and operations



Behavioral rules

Workflows and state machines

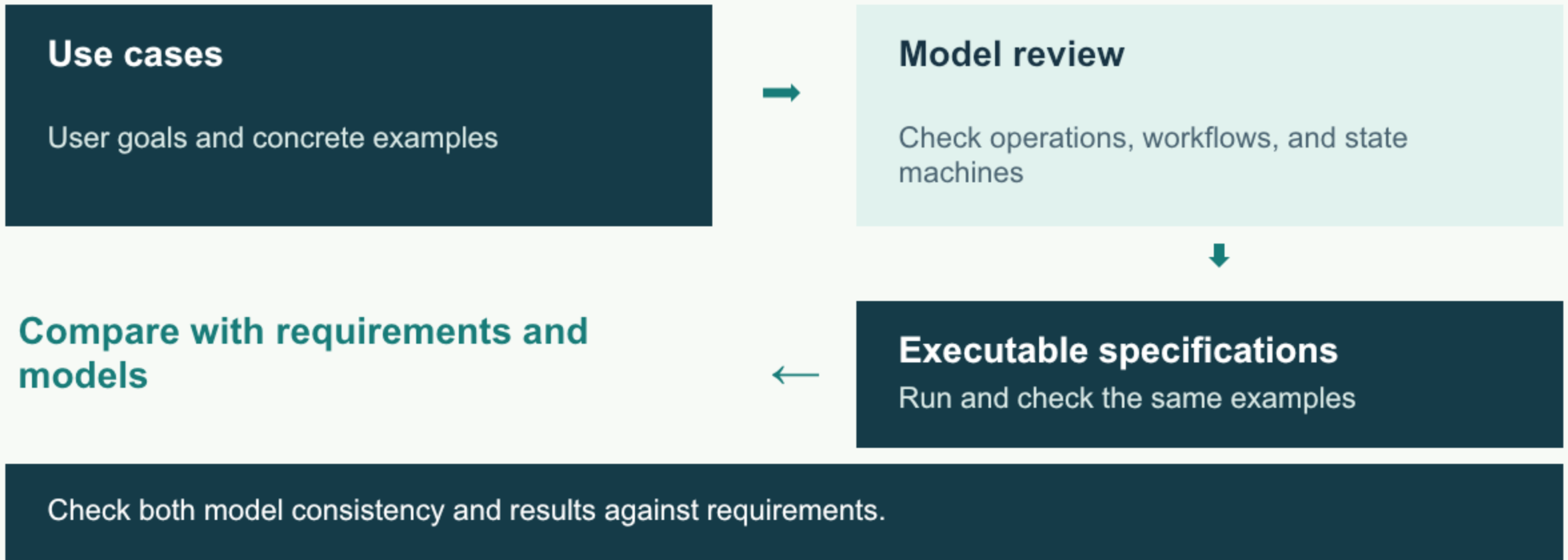


Direct execution

Less translation into code and comparison

Descriptions close to domain knowledge become the basis for checking and execution.

Requirements links and execution results



Shared rule ownership and evaluation

Domain ownership

Keep business rules together



Rule engine

Use a common evaluation mechanism

Remaining design issue

Map first-class rule models to execution

Shared ownership and evaluation reduce separate implementations of the same rule.

Bound changes, then check them with types

CBD public contracts

Component connections and conditions of use

Scala types

Consistency of values, inputs, and results

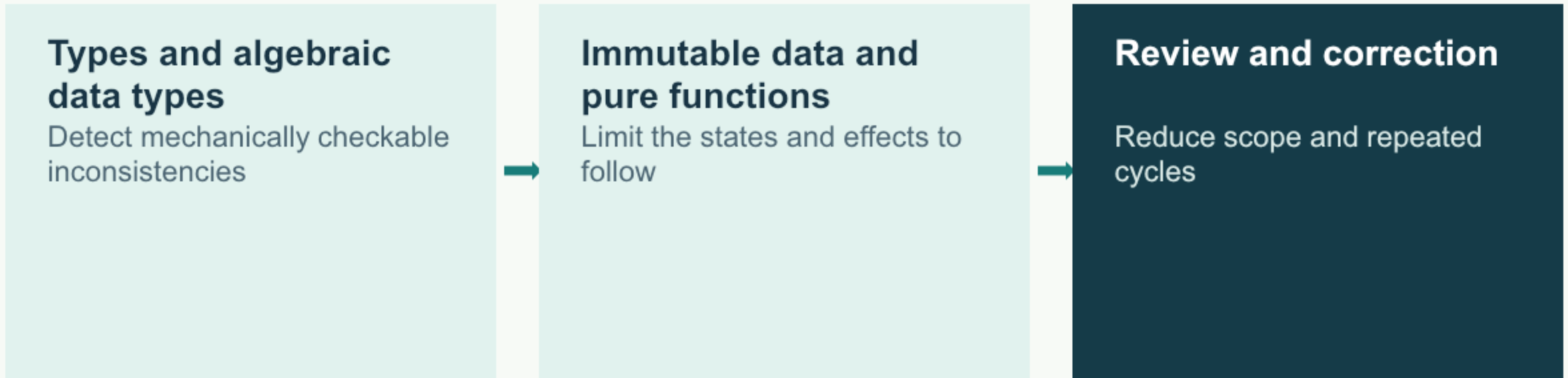


Contract-preserving changes

Focus checks on the component and connections

Preserving public contracts helps focus review on a component and its connections.

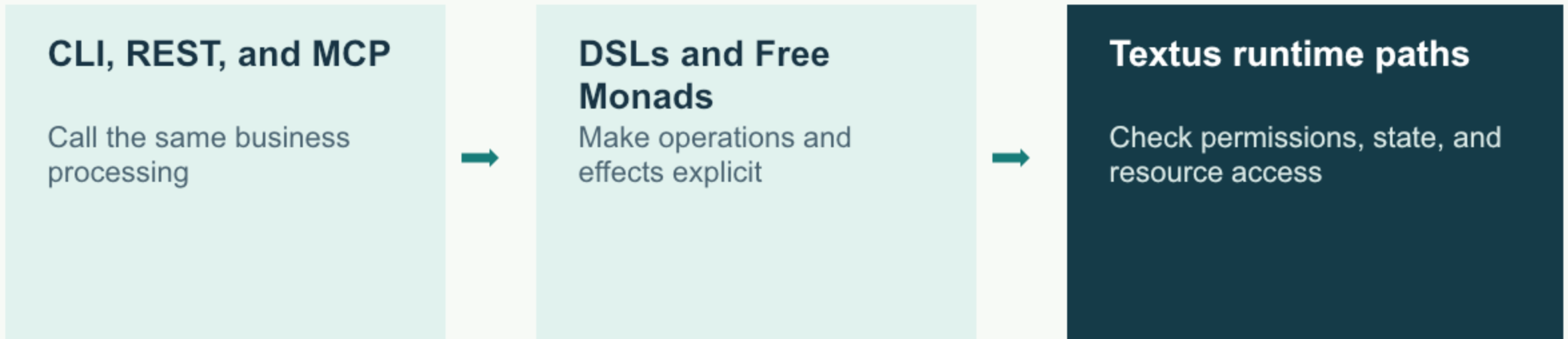
Reduce problems reaching AI review



Combine mechanical checks and limited state to reduce review and correction work.

Design effects; no measured reduction rate is claimed.

Permitted operations use shared runtime paths



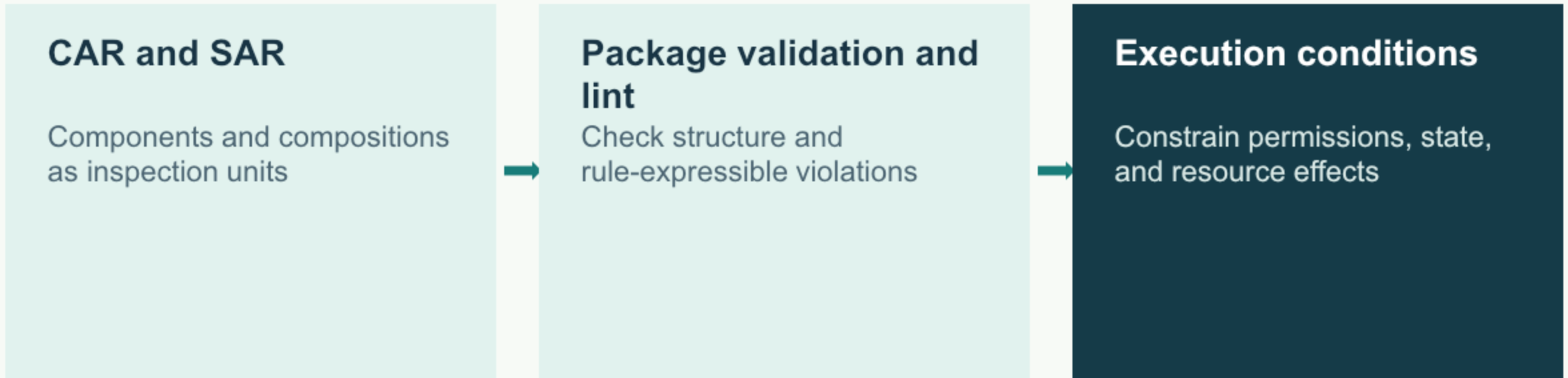
Different entry points use the same business processing and runtime constraints.

Multiple harnesses cover different conditions

Condition already checked	Further check still needed
A type-correct implementation	Business-rule conformity still needs checking
An operation matching the model	Check current state and user permissions
Requirements verification	Compare results with concrete examples

Other harnesses check conditions left open after one check passes.

Component checks and runtime constraints



Combine checks on distributed components with conditions on their use.

Scope and implementation status: Episode 7. Malicious code needs separate isolation.

Align knowledge with component contracts

BoK glossary

Connect literate models and object models



RDF and MCP

Supply knowledge with sources, versions, and scope



Skills

State change boundaries and responses to checks

Align models, contracts, and knowledge as premises for AI decisions.

The aim is to align knowledge with the target and its contracts.

Use execution evidence to locate corrections

Observation models

Map operations, transitions, and failures to models



Executable specifications

Compare required examples with actual results



Compare with requirements and models



AI review and human judgment

Assess meaning and requirements conformity

Map evidence to models to locate corrections and decisions that need review.

Models and implementation in order processing

Use cases

Buyer goals, stock shortages, and payment failures



Static models and rules

Orders, line items, amounts, and pricing rules



Workflows and state machines

Reserve stock, authorize payment, confirm

Connect requirements, structure, and behavior to inspect the order process.

Different conditions on the same order process

Situation	Additional constraint or verification
Types are correct	Check pricing and tax rules against examples
Confirmation is called	Check payment approval and user permissions
Payment fails	Prevent confirmation and apply defined compensation

Constrain the same process with different conditions to cover gaps between checks.

Series roles: Episodes 1–5

Episode 1: development conditions

Cost through verification, correction, and approval

Episodes 2–4: models

CML, model up/down, static and dynamic models

Episode 5: architecture

Roles and combinations that optimize the whole process

Connect the foundation of models as programs to the architecture.

Series roles: Episodes 6–10

Episodes 6–7: Scala and runtime

Types, immutable data, DSLs, Free Monads, runtime checks

Episode 8: knowledge

BoK, RDF, and MCP align premises and scope

Episodes 9–10: observation and verification

Evidence, executable specifications, AI review, human judgment

Later episodes explain each mechanism; Part 5 explains how they work together.

Architecture Harness

SimpleModeling reference architecture

Models as a common basis

CML: concepts and rules

Check requirements and realization



Generation and completion

Cozy: generate structure

CBD + Scala: boundaries and types



Shared execution rules

Textus: execute models directly

Constrain permissions and effects

Link knowledge, observation, and verification to the same models and contracts

BoK, RDF, MCP: align premises / Compare execution evidence with requirements examples

Use complementary harnesses to check conditions left open by individual checks

Correct types: check business rules / Valid operations: check permissions and state

Reduce custom implementation, review scope, and repeated AI review

Maintain required quality while aiming for faster development and lower AI costs