EXAMPLE 03

Coordinate two reusable libraries

An order release needs several capabilities to agree: calculate its value, check available stock, apply a limit and obtain a risk decision. This example assembles those responsibilities from two synthetic Java libraries and a small application output component.

2 libraries assembled · 7 nodes coordinated · 7 scenarios checked

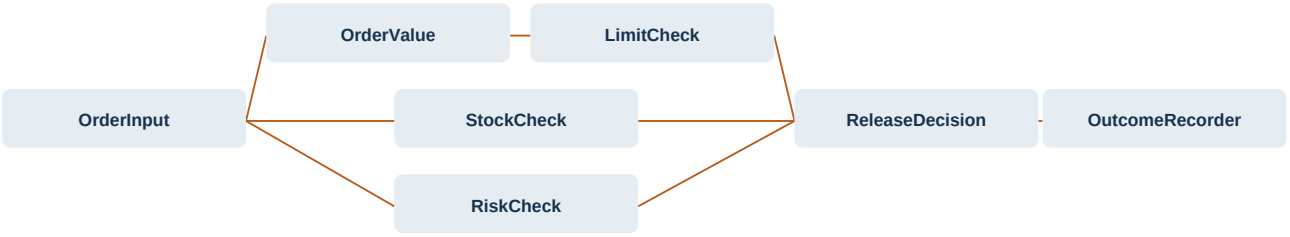


Illustration of the demonstrated workflow.

Observed result

The local run builds two separate JARs and executes freshly generated coordination across seven nodes. Normal and equality cases are accepted; limit and stock failures are rejected with the expected priority. A deliberately unavailable risk check raises an exception and produces no output. The next valid event succeeds. All seven scenarios match the expected outcomes.

Case or component	Recorded result
Normal / exact limit	ACCEPTED / ACCEPTED
Above limit / insufficient stock	REJECTED_LIMIT / REJECTED_STOCK
Both checks fail	REJECTED_LIMIT (defined priority)
Risk unavailable / next valid event	EXCEPTION, no output / ACCEPTED

Synthetic demonstration with recorded results, not a customer case study.

Follow the five-step recorded workflow

Download the matching investigation bundle

To reopen the investigation in the Analyser, save the HTML reel and bundle in the same folder.

FOLLOW THE DEMONSTRATION

5-stage walkthrough

1. Define the joined decision

An order needs acceptable value, available stock and a successful risk check before release.

2. Package the value checks

The first library provides the order input, amount calculation and stock check.

3. Package the policy checks

The second library provides the limit, risk and release decision. Both libraries are synthetic demo components.

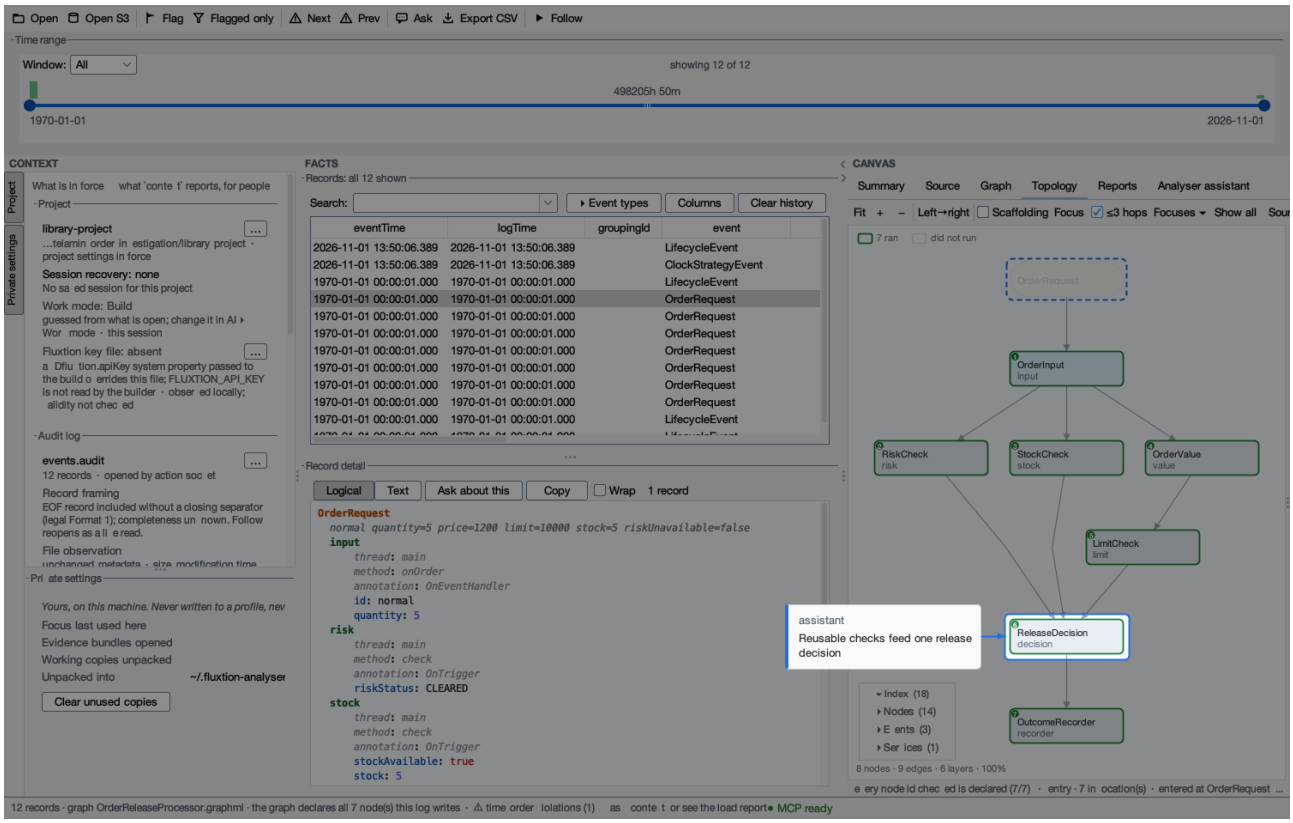
4. Compile their coordination

Fluxtion connects the seven-node graph; the application adds an output recorder and builds against the two JARs.

5. Check failures and recovery

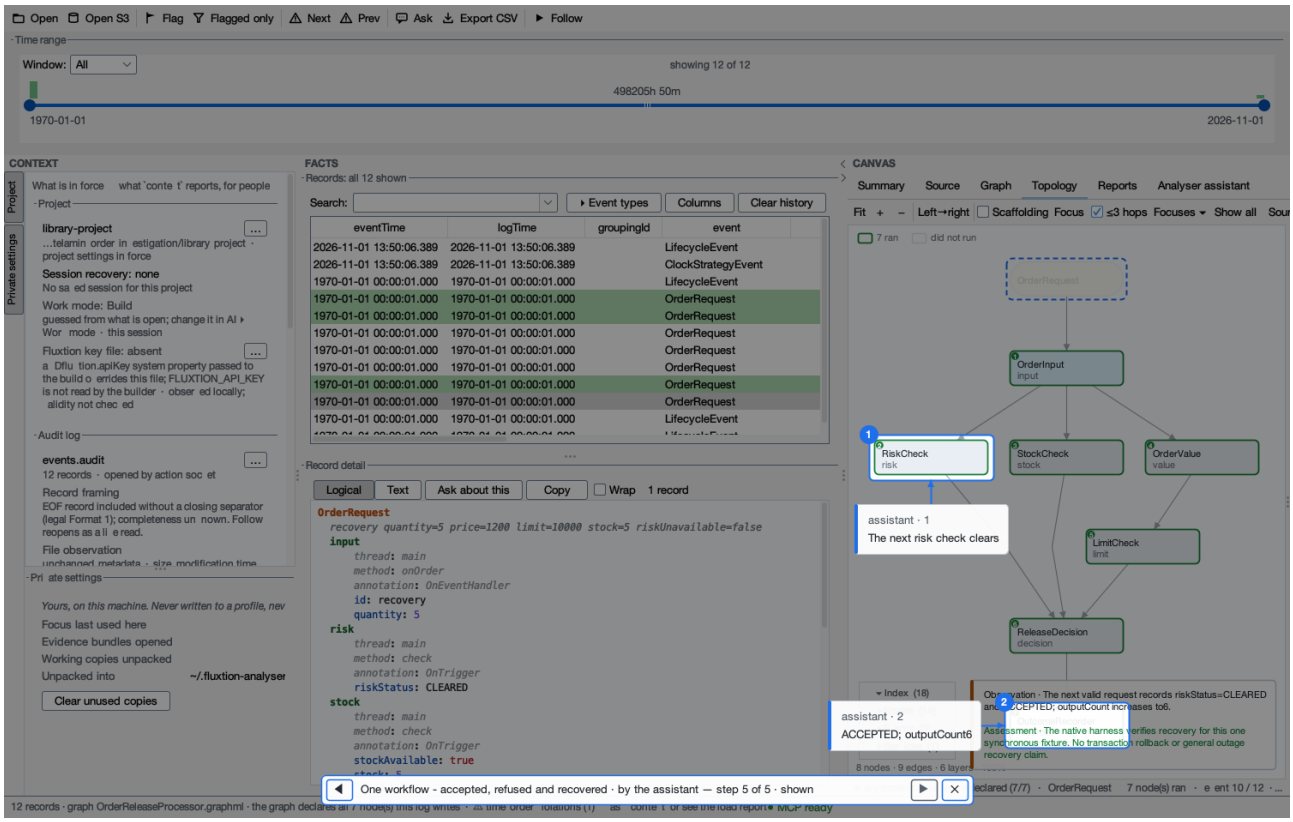
Seven scenarios pass. The synthetic risk exception produces no output; the next valid order succeeds.

The new application graph includes the order event and seven processing nodes.



FOLLOW THE RECORDED OUTCOMES

The next valid order is accepted after the risk-check exception.



The new workflow produces the expected accepted and rejected decisions. The native harness confirms that the unavailable risk check produces no output. The following valid request is accepted, taking the recorded output count to six. The saved investigation and reel preserve the graph and five observed paths for review.

An evidence bundle is a portable investigation created by the Analyser from selected records, the graph and a saved walkthrough. A person and their AI assistant guide the investigation. The bundle can be reopened; the reel and PDF are for reading and sharing.

What is an evidence bundle?

Why this matters

The calculation and policy capabilities stay separately packaged while the application defines their relationships. The checks make rejection priority and one exception path visible, so reuse does not hide how the combined workflow behaves.

Apply it to your process

Bring two real libraries or in-house capabilities whose coordination is costly to maintain. A pilot can assess adapters, dependency conflicts, exception paths and what the combined system records. Compatibility and licensing must be established for those specific libraries.

Discuss a five-stage pilot

TECHNICAL APPENDIX

Reproduce this result

Unzip the two-library reproduction package, then run from its library-workflow directory with the separately supplied reviewed runtime:

```
python3 run.py \
  --runtime /path/to/fluxtion-runtime-1.1.3.jar \
  --out results
```

The runner builds two synthetic JARs and the included fresh generated Java, then checks seven scenarios. It verifies the runtime hash and records portable javac, jar and java commands. No generation, network or model call is needed.

Source and reproduction package for this example

Compiler 1.0.83 generated this workflow locally. The owner-authorized Apache 2.0 notice was applied without changing the Java package/body, then the included processor was rebuilt and all seven scenarios passed.

Runtime prerequisite: com.telamin.fluxtion:fluxtion-runtime:1.1.3. Download the JAR without an account; the runner checks its SHA-256. The source ZIP includes RUNTIME-PREREQUISITES.txt with the complete download command.

Download runtime JAR | Published POM: declares GNU AGPL version 3, separate from Apache 2.0 demo code.

SHA-256: 2bd45eb62409403422d5a676a7ad177f1984aa80dc399c20ceb6b467eab4b775

Detailed evidence

decisions.csv

commands.json

manifest.json

events.audit

summary.txt

Open this workflow's Analyser evidence bundle

Recorded investigation method and reopen instructions

Actual investigation tool requests and responses

All exported PDFs, screenshots and verification receipts

Scope and interpretation

Both JARs are synthetic; this demonstrates component boundaries and coordination, not compatibility with arbitrary vendor libraries.

The synthetic risk exception tests one in-process failure and subsequent event. No transaction rollback, network recovery, dependency conflict, throughput or production readiness is established.

This is a new tested workflow, not a before-and-after repair. Seven processing nodes and an additional order-event vertex appear in the application graph. Its branches are dependencies, not parallel execution threads.

The actual Analyser capture used its isolated local REST action socket with hosted OpenAI inference, not MCP transport or a local model. Its unsigned bundle was verified and reopened on the same machine. Source roots, flags and replay payload do not travel in the bundle.

The audit contains 12 records and all seven logged node IDs match the supplied graph. The native harness separately checks the synthetic exception and unchanged output count; absence of an audit output alone would not prove non-execution. Initialization wall time precedes a fixed fixture clock, so the investigation follows record indexes and values, not wall-time claims.

Demo code is Apache 2.0 under owner authorization dated 1 November 2026. Runtime, third-party and general customer-output terms remain separate. Source and artifact hashes compare bytes; they do not authenticate a sender or certify correctness.

Deployment and data flows | Applicable product rights