

Drools Expert-Level Guide (Enhanced Edition)

From Rule Author to Decision Platform Architect

1. Mental Model Upgrade: How Experts Think About Drools

Beginner Thinking

“Rules are if–then logic executed by an engine.”

Expert Thinking

“Rules are **versioned, composable decision assets** executed by a **policy-aware inference engine** that must be **auditable, explainable, and region-configurable**.”

Key mindset shifts:

From	To
Rule files	Rule products
DRL logic	Decision intent
Stateless execution	Context-aware decisioning
Single ruleset	Layered rule groups
Local dev	Globally deployed decision services

2. Advanced Rule Organization: Rulesets, Rule Groups, and Decision Layers

2.1 Ruleset ≠ Rule File

Ruleset (Expert Definition):

A versioned, deployable collection of rules that fulfills **one business decision responsibility**.

Examples:

- TravelDelayEligibilityRuleset
- AutoBISettlementRuleset
- FraudPreScreenRuleset

Each ruleset should:

- Have a **clear decision contract**
 - Be independently versioned
 - Produce a **decision outcome**, not just side effects
-

2.2 Rule Groups (Salience-Free Design)

Avoid heavy reliance on `salience`. Instead, structure **explicit execution stages**.

Recommended Rule Group Taxonomy

```
rules/
  └── 01_validation/
  └── 02_normalization/
  └── 03_eligibility/
  └── 04_calculation/
  └── 05_exceptions/
  └── 06_enrichment/
  └── 07_final_decision/
```

Each folder:

- Maps to a **decision phase**
- Has clear entry/exit assumptions
- Is testable in isolation

💡 **Rule groups are architectural boundaries**, not just folders.

2.3 Agenda Groups vs Ruleflow Groups

Feature	Use Case
<code>agenda-group</code>	Simple sequencing
<code>ruleflow-group</code>	BPMN / orchestration
Stateless sessions	High-volume STP
Stateful sessions	Investigations / long-lived cases

Expert Rule:

👉 Prefer `ruleflow-group` when rules map to **business process stages**.

3. Rule Templates: The Key to Global Reusability

3.1 Why Templates Matter at Scale

Without templates:

- Rules explode by region
- Logic gets duplicated
- Changes become ungovernable

With templates:

- Logic = **stable**
 - Data = **configurable**
 - Regions = **data-driven**
-

3.2 Template Design Principles

Bad Template

Rule = Logic + Constants

Good Template

Rule = Logic + Parameters + Metadata

3.3 Canonical Template Example (Insurance-Grade)

```
rule "Eligibility - #{@coverageCode} - #{@region}"
agenda-group "eligibility"
when
  Claim(
    coverageCode == "#{coverageCode}",
    lossRegion == "#{region}",
    lossAmount >= #{@minAmount},
    lossAmount <= #{@maxAmount}
  )
then
  decision.markEligible(
    "#{decisionCode}",
    "#{reasonCode}"
  );
end
```

CSV Example

```
coverageCode, region, minAmount, maxAmount, decisionCode, reasonCode  
TRAVEL_DELAY, EMEA, 300, 5000, ELIGIBLE, TD_EMEA_STD  
TRAVEL_DELAY, APAC, 200, 3000, ELIGIBLE, TD_APAC_STD
```

3.4 Template Governance Strategy

Layer	Ownership
Template logic	Platform / Architecture
Parameter values	Business / Product
CSV storage	Config repo or DB
Approval	Decision governance workflow

4. Global Deployment Best Practices (Critical)

4.1 Region-Aware Rule Design

Never hardcode geography.

Use:

- regionCode
- jurisdiction
- regulatoryContext

```
when  
    Claim(region == Region.EMEA)
```

Instead of:

```
when  
    Claim(country == "Germany")
```

4.2 Rule Inheritance via Composition (Not Copy-Paste)

Base Ruleset

- Global defaults

Overlay Ruleset

- Region-specific deltas

Execution model:

4.3 Versioning Strategy (Non-Negotiable)

Asset	Versioned?
DRL	✓
Templates	✓
CSVs	✓
Decision outcome	✓

Semantic versioning recommended

```
TravelDelayEligibilityRuleset
└── 1.3.0 (logic change)
└── 1.3.1 (threshold update)
```

5. Reusability Patterns (Expert-Level)

5.1 Rule as Capability, Not as Flow

Bad:

“This rule only works in FNOL”

Good:

“This rule evaluates eligibility regardless of entry point”

Rules should be:

- Channel-agnostic
 - System-agnostic
 - Reusable by FNOL, Adjuster UI, Simulation, AI agents
-

5.2 Decision Object Pattern

Avoid mutating the Claim directly.

```
Decision decision = new Decision();
```

Rules write to `Decision`, not to domain objects.

Benefits:

- Explainability
 - Replayability
 - Simulation support
-

6. Explainability & Audit (Enterprise-Critical)

6.1 Structured Decision Output

Each rule must emit:

- Rule ID
- Rule version
- Reason code
- Confidence / severity (optional)

```
decision.addEvidence(  
    "ELIGIBILITY_RULE_102",  
    "TD_AMOUNT_THRESHOLD_MET",  
    Severity.HIGH  
) ;
```

6.2 Traceability for Regulators

Store:

- Input snapshot
- Fired rules
- Output decision
- Rule version hash

This enables:

- Regulatory audits
 - Claim disputes
 - Model drift analysis
-

7. Performance & Scaling

7.1 Stateless Session Best Practices

- One request → one session
 - No globals with mutable state
 - No static caches inside rules
-

7.2 Fact Design for Performance

Bad	Good
Large domain objects	Lean projection facts
Nested conditions	Precomputed flags
String comparisons	Enums / ints

8. Testing Strategy (Beyond Unit Tests)

8.1 Rule Unit Tests

- One rule
- One scenario
- One expected outcome

8.2 Ruleset Scenario Tests

- End-to-end decision
- Golden datasets

8.3 Regression Packs

- Historical claims replayed
 - Required for every deployment
-

9. CI/CD & Promotion (Enterprise-Grade)

9.1 Pipeline Stages

Validate DRL
→ Compile Rules
→ Run Scenario Tests
→ Package KJAR

→ Deploy to Artifact Repo
→ Promote to Env

9.2 Environment Strategy

Env	Purpose
DEV	Rule authoring
SIT	Integration
UAT	Business validation
PROD	Locked logic

10. Anti-Patterns (What Experts Avoid)

- ✗ Hardcoded values
 - ✗ Copy-pasted rules
 - ✗ Salience wars
 - ✗ Rule logic tied to UI
 - ✗ No versioning
 - ✗ No explainability
-

11. Final Expert Checklist

Before calling yourself **Drools Expert**, you should be able to:

- Design reusable rule templates
- Support multi-region deployments
- Explain every decision to an auditor
- Version and promote rules safely
- Simulate decisions at scale
- Integrate rules with AI-generated inputs
- Treat rules as **products**, not scripts

Part 2

Drools for Insurance STP

An Expert-Level Guide to Decision Automation at Enterprise Scale

1. Purpose of This Document

This document defines **how Drools should be designed, structured, governed, tested, deployed, and operated** as the **core deterministic decision engine** in an **insurance Straight-Through Processing (STP)** platform.

It is intended for:

- Platform architects
- Senior engineers
- Rules engineers
- Claims and underwriting decision designers
- AI / decision automation teams

The focus is **not just “how to write rules”**, but **how to build a sustainable, global decision platform** using Drools.

2. Role of Drools in Insurance STP

2.1 Why Drools Exists in STP

In insurance STP, Drools is used to:

- Make **deterministic, explainable decisions**
- Enforce **regulatory and contractual rules**
- Enable **consistent outcomes across channels**
- Support **high-volume, low-latency automation**

Drools is **not**:

- A workflow engine
- A UI rules engine
- A replacement for AI/ML

- A data store

Drools **is**:

A deterministic decision engine that transforms **facts** → **decisions** in a way that is **auditable, repeatable, and legally defensible**.

2.2 Typical STP Decisions Implemented in Drools

Category	Examples
Eligibility	Is this claim eligible for coverage?
Thresholds	Is loss amount within auto-settlement limits?
Routing	Can this be STP or must it go to an adjuster?
Compliance	Does jurisdiction require manual review?
Exception handling	Does any exclusion apply?
Calculations	Payout caps, deductibles, waiting periods
Pre-fraud screening	Deterministic fraud indicators

3. Core Design Principles (Non-Negotiable)

3.1 Determinism Over Intelligence

Drools decisions **must always**:

- Produce the same output for the same input
- Be explainable without probabilistic reasoning
- Be replayable months or years later

AI may **assist** Drools (e.g., classification, extraction), but **Drools owns the final deterministic decision.**

3.2 Rules Are Products, Not Scripts

Every ruleset should be treated as:

- A versioned product
- With a defined contract
- With owners
- With a lifecycle

If you cannot answer:

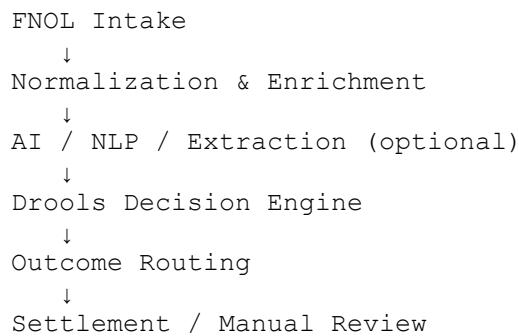
- What decision does this ruleset own?
- Who owns it?
- How is it tested?
- How is it deployed?

Then the ruleset is not production-ready.

4. STP Decision Architecture with Drools

4.1 Position in the STP Flow

Typical STP flow:



Drools **must not**:

- Call external systems
- Depend on runtime IO
- Perform orchestration

Drools **only evaluates facts** already provided.

5. Ruleset Architecture

5.1 What Is a Ruleset?

A ruleset is a **cohesive collection of rules** that:

- Owns a **single business decision**
- Produces a **clear outcome**

- Can be versioned independently

Examples:

- TravelDelayEligibilityRuleset
 - AutoGlassSTPDecisionRuleset
 - HealthClaimPreAuthRuleset
-

5.2 Ruleset Boundaries

A ruleset:

- Must not span multiple unrelated decisions
- Must not depend on execution order of other rulesets
- Must be callable as a **pure function**

Decision = evaluate(Facts)

6. Rule Grouping and Execution Phases

6.1 Execution Phases (Recommended)

Rules should be grouped by **decision phase**, not by feature.

```
rules/
└── 01_validation/
└── 02_normalization/
└── 03_eligibility/
└── 04_calculation/
└── 05_exclusions/
└── 06_routing/
└── 07_final_decision/
```

Each phase:

- Has a single responsibility
 - Can be tested independently
 - Can be reasoned about by business users
-

6.2 Agenda Groups vs Ruleflow Groups

Feature	Usage
agenda-group	Simple sequencing
ruleflow-group	BPMN / orchestration
Stateless session	High-volume STP
Stateful session	Long-running investigations

Best Practice

- Use `agenda-group` for most STP
 - Introduce `ruleflow-group` only when mapping directly to business workflows
-

7. Global Deployment Strategy

7.1 Region-Aware Design

Never hardcode:

- Country names
- Jurisdiction logic
- Regulatory rules

Always externalize:

- Region
- Jurisdiction
- Regulatory context

when

```
Claim(regulatoryContext == RegulatoryContext.EMEA)
```

7.2 Layered Rule Composition

Use **overlay patterns**, not duplication:

```
Global Rules
  ↓
Regional Rules
  ↓
Product Rules
  ↓
Exception Rules
```

This enables:

- Global consistency
 - Local flexibility
 - Controlled variance
-

8. Rule Templates (Critical for STP Scale)

8.1 Why Templates Are Mandatory

Without templates:

- Rules explode combinatorially
- Maintenance becomes impossible
- Regional rollout slows dramatically

Templates allow:

- One logic definition
 - Thousands of parameterized rules
-

8.2 Template Design Principles

A template should:

- Contain **only logic**
- Accept **all variability as parameters**
- Be owned by the platform team

Bad:

Logic + values in DRL

Good:

Logic in DRL template, values in CSV / config

8.3 Insurance-Grade Template Example

```
rule "Eligibility - @{coverageCode} - @{region}"  
agenda-group "eligibility"  
when  
    Claim(  
        coverageCode == "@{coverageCode}",
```

```

        lossRegion == "@{region}",
        lossAmount >= @{minAmount},
        lossAmount <= @{maxAmount}
    )
then
    decision.markEligible(
        "@{decisionCode}",
        "@{reasonCode}"
    );
end

```

9. Decision Output Model (Mandatory)

9.1 Never Mutate Domain Objects

Rules must **not** directly change Claim, Policy, or Coverage.

Instead, write to a **Decision object**.

```
Decision decision = new Decision();
```

Benefits:

- Explainability
- Replayability
- Auditability
- Simulation support

9.2 Decision Evidence

Each rule should emit structured evidence:

```
decision.addEvidence(
    ruleId,
    ruleVersion,
    reasonCode,
    Severity.HIGH
);
```

This enables:

- Claim dispute resolution
- Regulatory audits
- AI explainability overlays

10. Explainability & Auditability

10.1 Required Audit Artifacts

For every STP decision, persist:

- Input fact snapshot
- Fired rules
- Rule versions
- Decision output
- Timestamp and environment

This is **non-optional** in regulated insurance environments.

11. Performance & Scale

11.1 Stateless Session Best Practices

- One request → one session
- No mutable globals
- No static caches
- No external calls

11.2 Fact Optimization

Avoid	Prefer
Large domain graphs	Lean projection facts
Strings	Enums
Deep nesting	Precomputed flags

12. Testing Strategy

12.1 Rule Unit Tests

- One rule
- One scenario
- One expected outcome

12.2 Ruleset Scenario Tests

- End-to-end decision
- Golden datasets

12.3 Regression Packs

- Replay historical claims
 - Mandatory for every release
-

13. CI/CD & Promotion Model

13.1 Pipeline Stages

DRL Validation
→ Compilation
→ Scenario Tests
→ Package (KJAR)
→ Artifact Repo
→ Promote by Environment

13.2 Environment Strategy

Environment	Purpose
DEV	Rule authoring
SIT	Integration
UAT	Business sign-off
PROD	Locked, audited

14. Governance Model

14.1 Ownership

Asset	Owner
Templates	Platform
Rulesets	Product / Business
CSV Parameters	Business
Promotion	Architecture / Ops

14.2 Approval Flow

1. Rule authored
 2. Scenario tests pass
 3. Business approves
 4. Architecture signs off
 5. Promote
-

15. Anti-Patterns (What Breaks STP)

- ✗ Hardcoded thresholds
 - ✗ Copy-paste rules
 - ✗ Salience wars
 - ✗ Channel-specific rules
 - ✗ No versioning
 - ✗ No audit trail
-

16. Final Expert Checklist

You are using Drools correctly for STP if:

- Every decision is explainable
 - Rules are region-agnostic
 - Templates drive scale
 - Rulesets are versioned products
 - Decisions can be replayed years later
 - AI augments but never replaces determinism
-

17. How This Fits with AI & Future Platforms

Drools:

- Enforces deterministic truth
- Provides explainable decisions
- Acts as the **last line of defense**

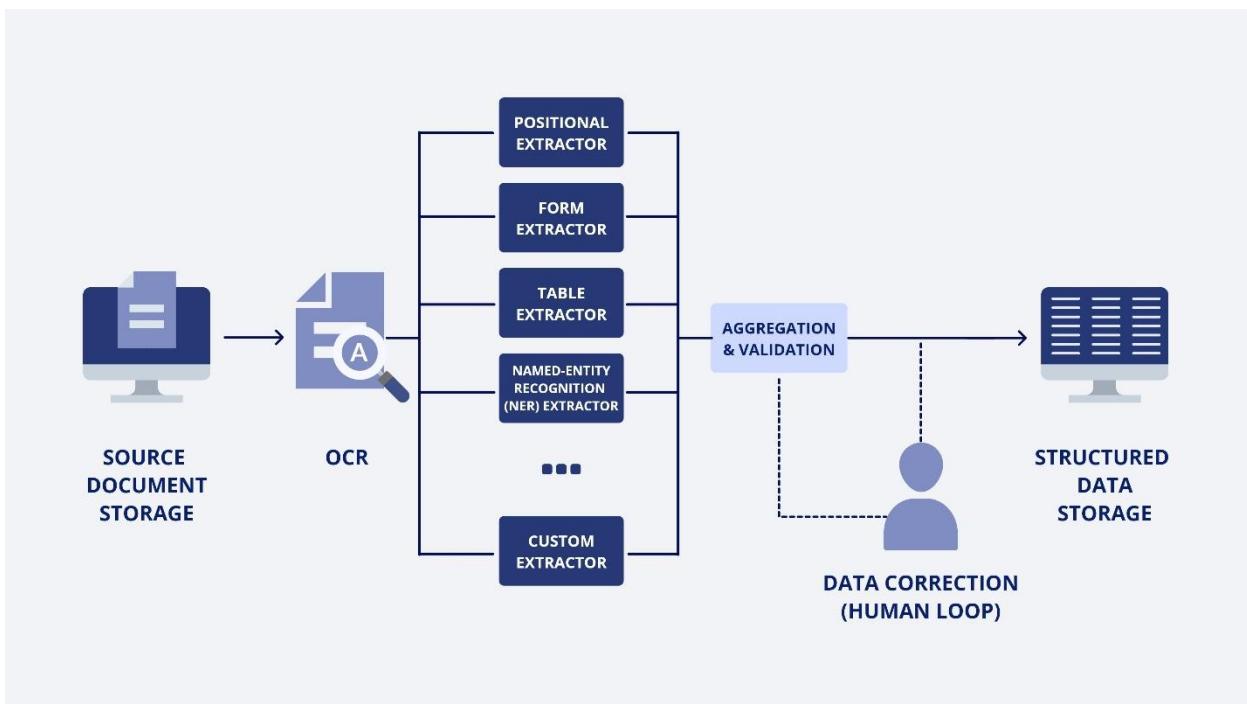
AI:

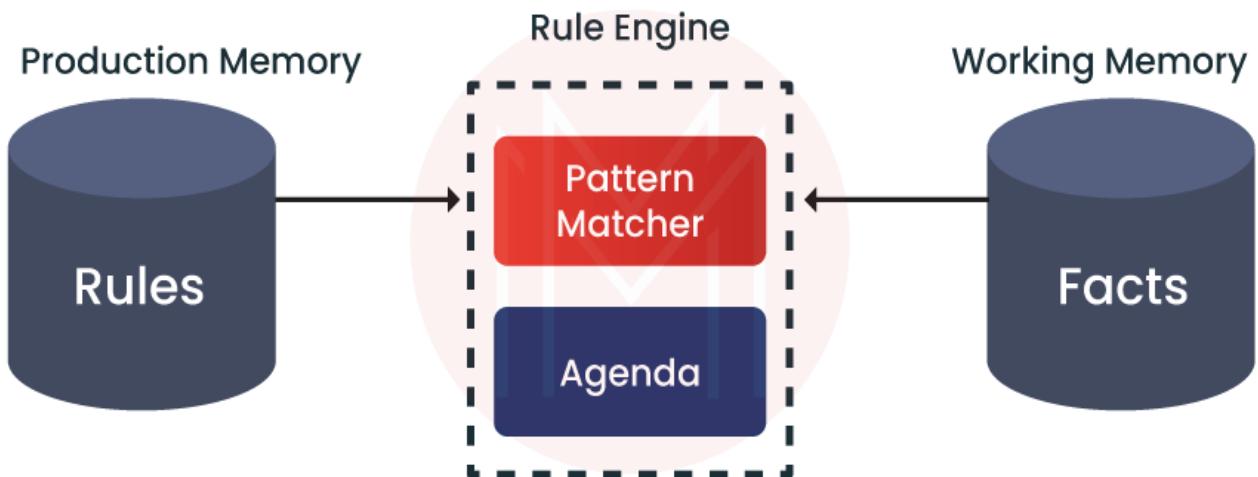
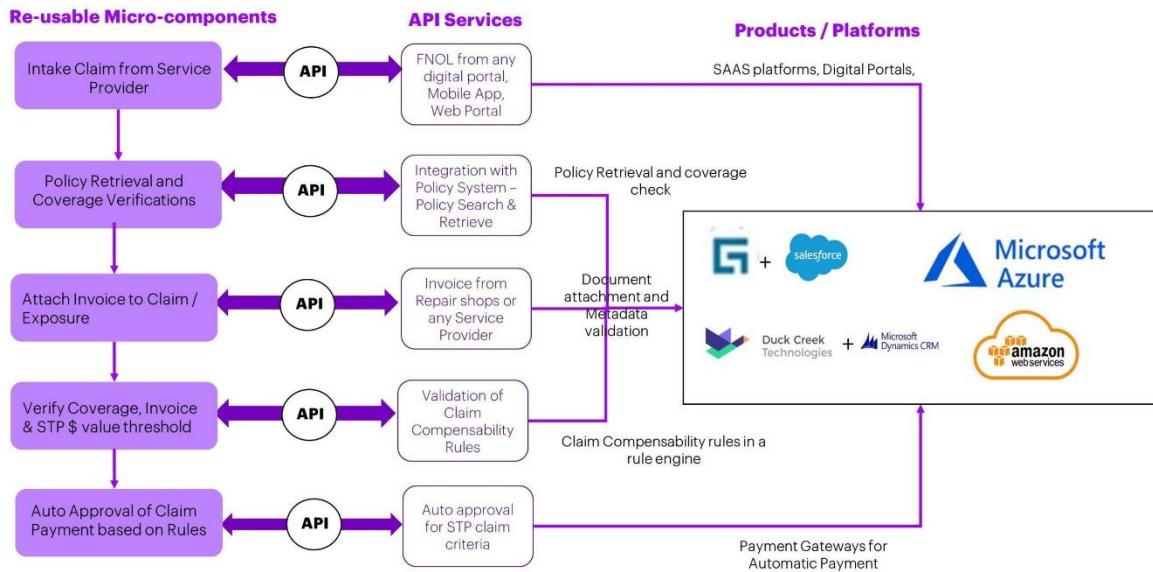
- Extracts data
- Classifies intent
- Suggests parameters
- Assists rule authoring

Drools remains the legal decision authority.

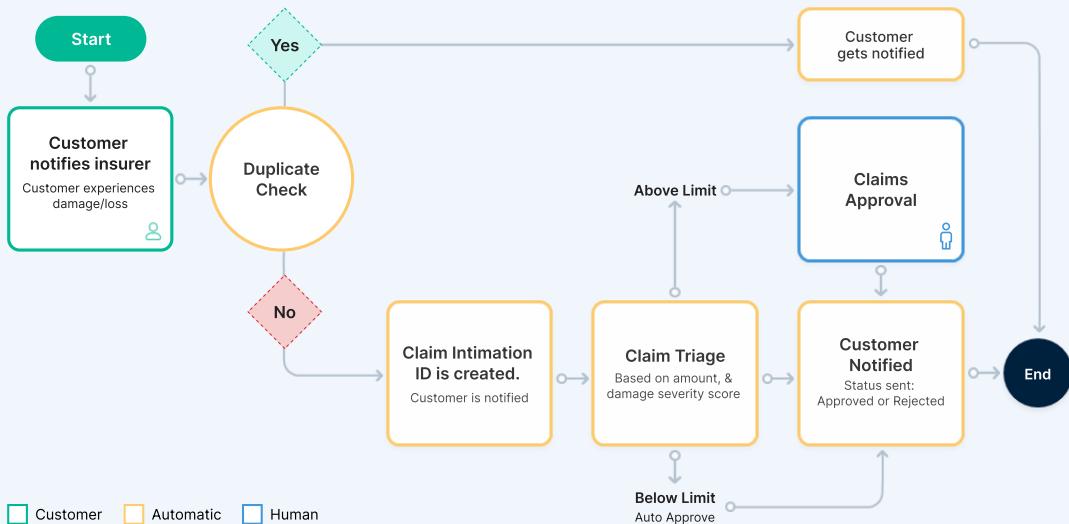
18. End-to-End Drools STP Architecture Diagrams

18.1 High-Level Insurance STP Decision Architecture





How AI transforms FNOL Automation



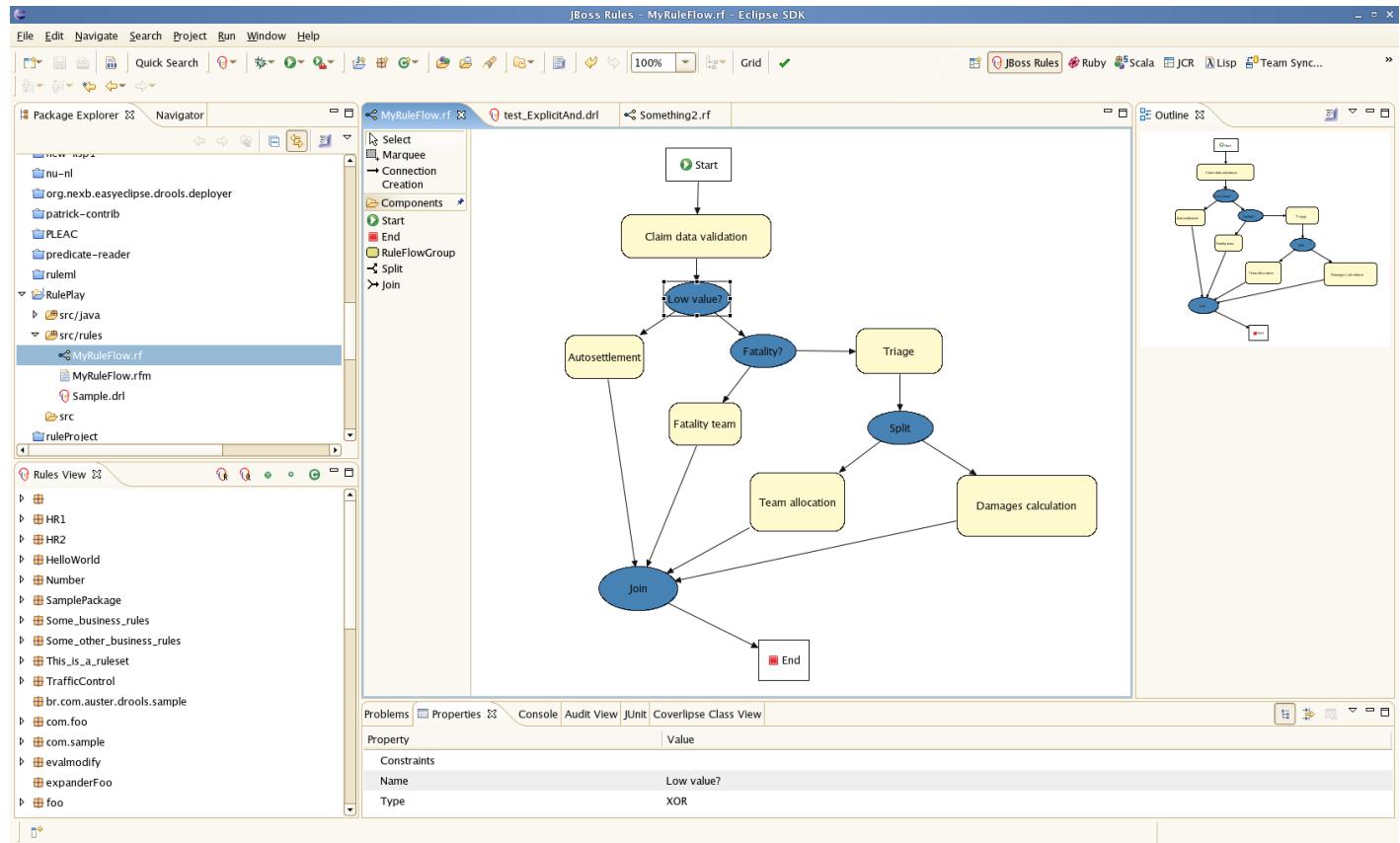
Narrative Walkthrough

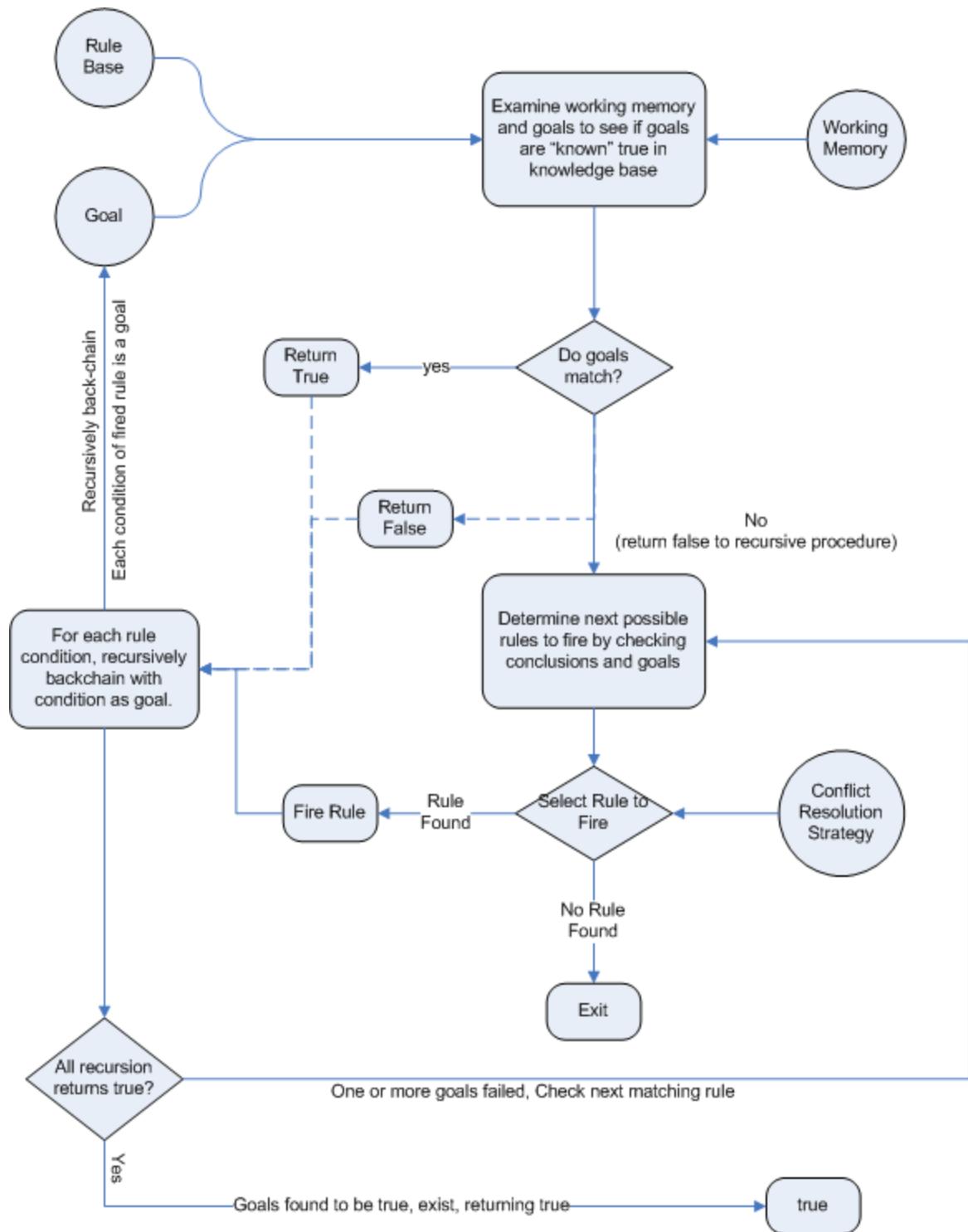
Channel (UI / API / Voice / Partner)
 ↓
 FNOL Intake API
 ↓
 Normalization & Canonical Mapping
 ↓
 AI / NLP / Extraction (optional, non-deterministic)
 ↓
 Drools Decision Engine (deterministic)
 ↓
 Decision Outcome
 ↓
 Settlement | Routing | Manual Review

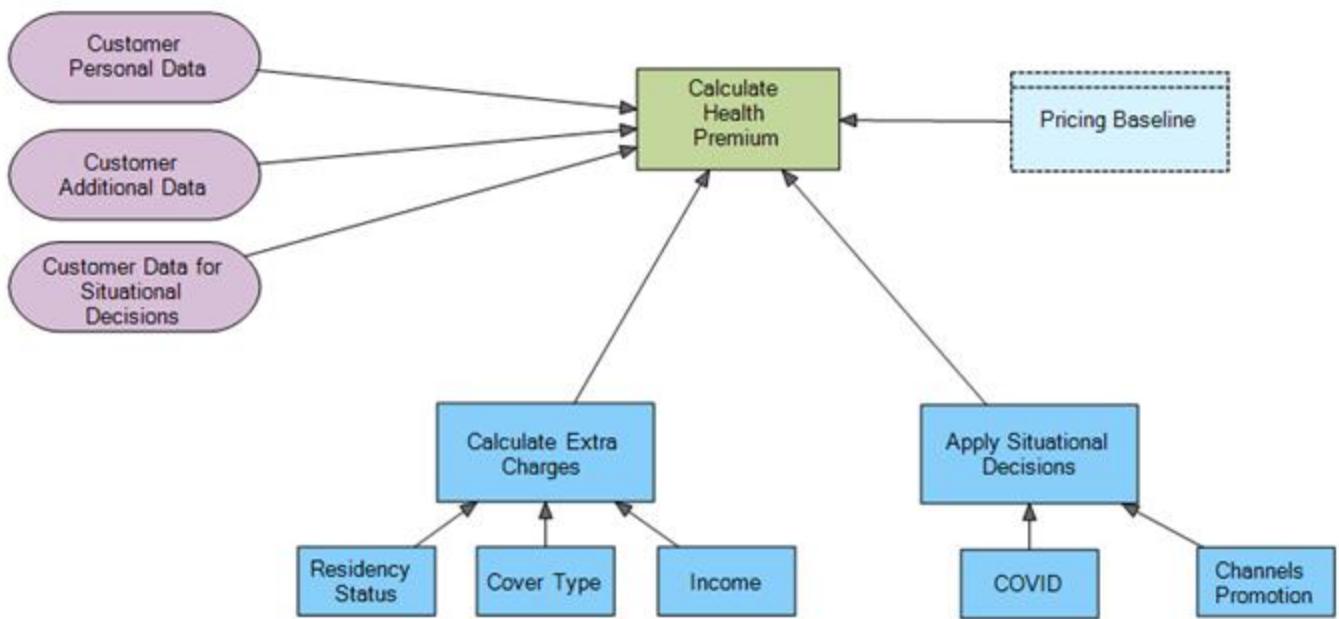
Key architectural rules:

- Drools never depends on channels
- Drools never calls external services
- Drools consumes **facts**, not raw payloads
- AI enriches facts but never decides

18.2 Detailed Decision Flow with Drools Rule Groups







©2021 FlexRule Pty Ltd. All rights reserved.

Execution Stages (Rule Groups)

- [01 Validation]
- ↓
- [02 Normalization]
- ↓
- [03 Eligibility]
- ↓
- [04 Calculation]
- ↓
- [05 Exclusions]
- ↓
- [06 Routing]
- ↓
- [07 Final Decision]

Each stage:

- Has a **single responsibility**
- Produces structured decision evidence
- Can short-circuit the flow if required

18.3 Global Deployment & Overlay Model

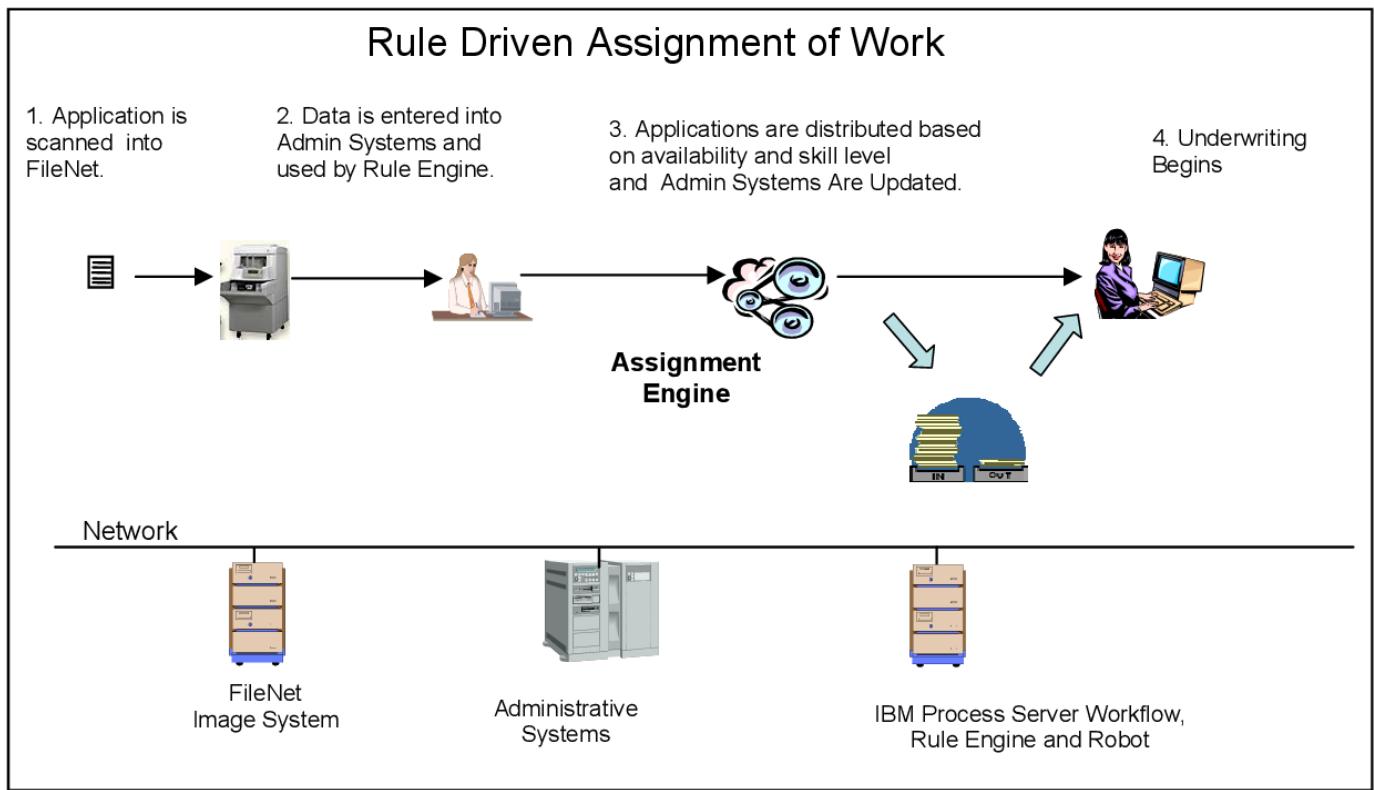
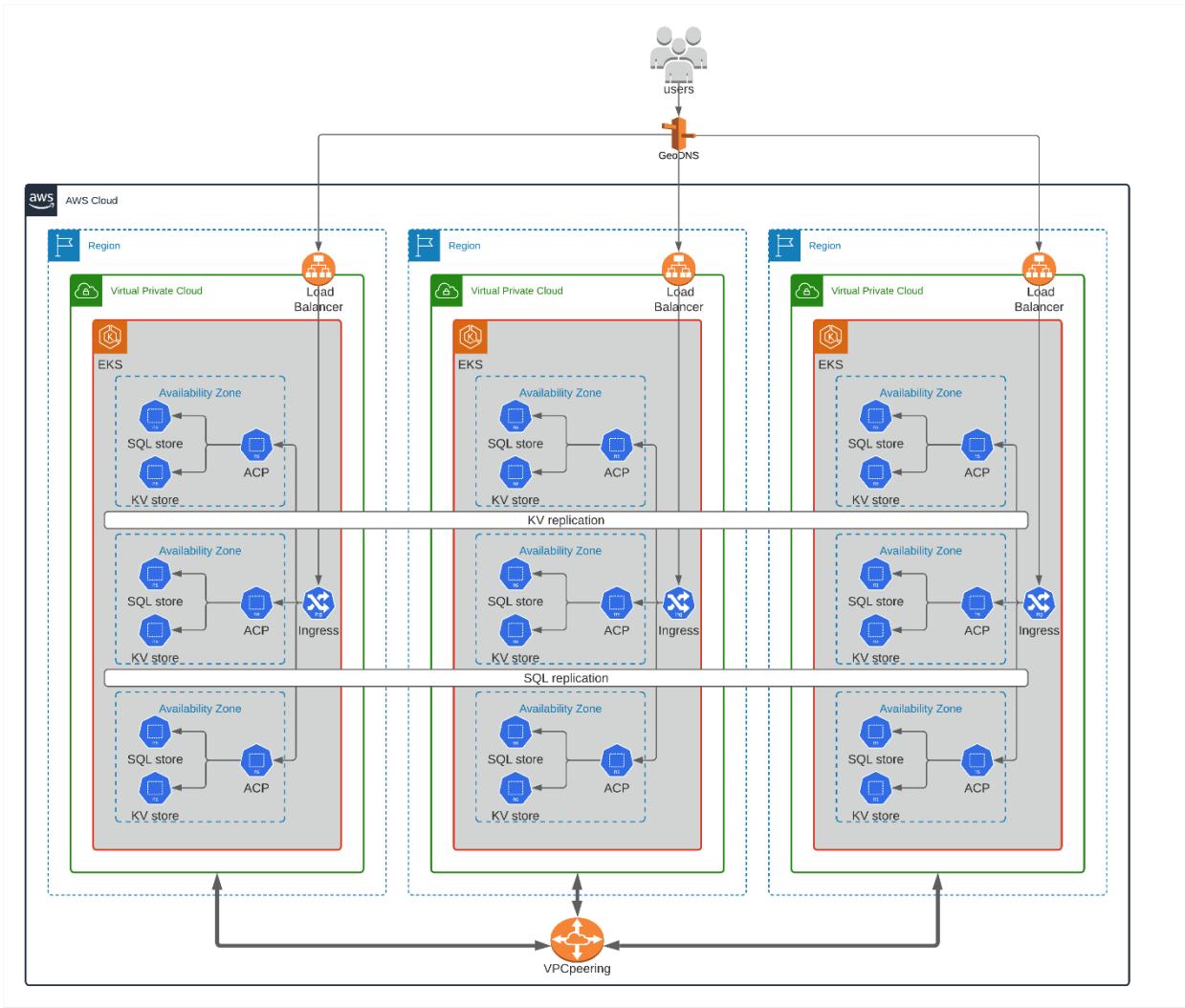


Figure 2





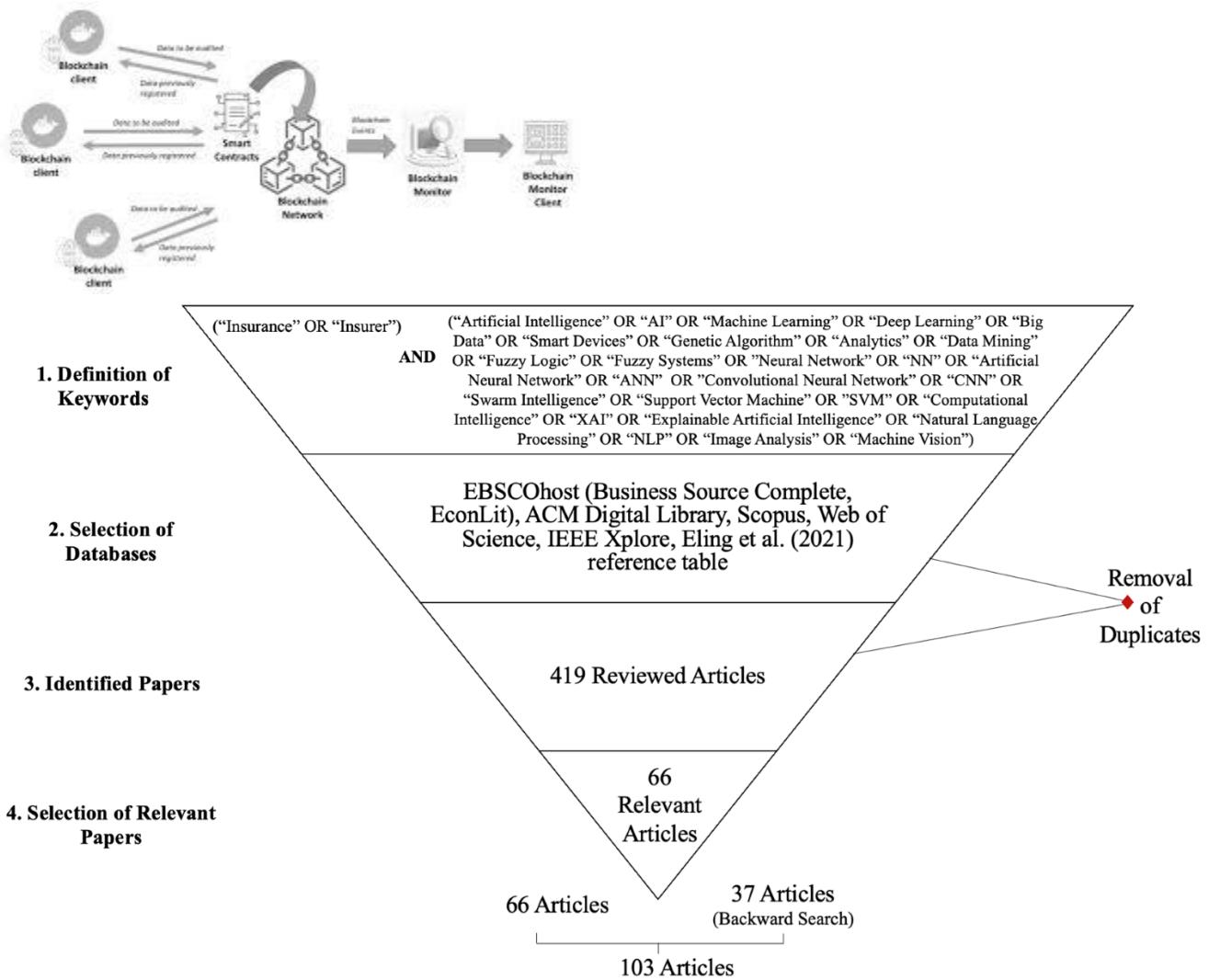
Overlay Strategy (Critical for Global STP)

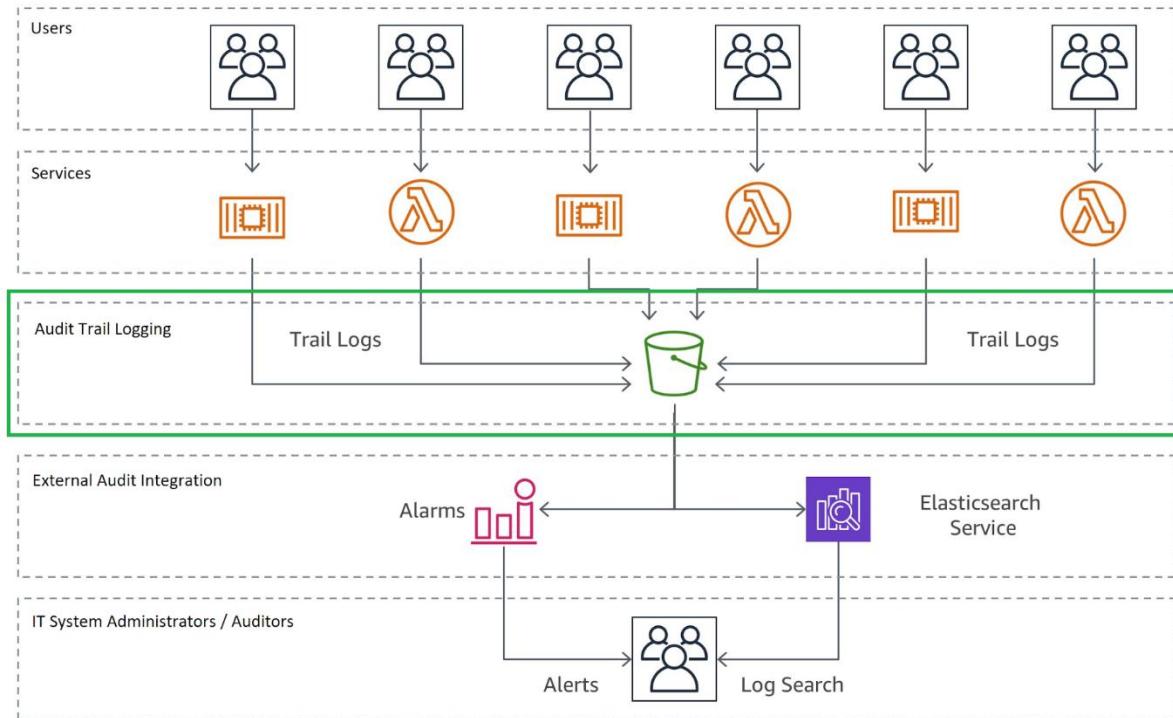
```
Global Rules (Base)
  ↓
Regional Rules (EMEA / APAC / LATAM)
  ↓
Product Rules (Travel / Auto / Health)
  ↓
Exception Rules (Regulatory / Carrier-specific)
```

Why this matters

- Global consistency
 - Regional compliance
 - Minimal duplication
 - Safe evolution
-

18.4 Decision Evidence & Audit Trail Flow





Audit Artifacts Persisted Per Decision

- Input fact snapshot
- Fired rule IDs
- Rule versions
- Decision outcomes
- Reason codes
- Environment & timestamp

This supports:

- Regulatory audits
- Claim disputes
- Model governance
- AI explainability overlays

19. Drools for Insurance STP – Certification Path

This is designed as an **internal enterprise certification**, not a vendor cert.
Think: “*Drools Black Belt for Insurance Decisioning.*”

Level 1 – Drools Foundations (Engineer)

Audience

- Software engineers
- Claims / underwriting technologists
- New rules authors

Competencies

- Drools fundamentals
- DRL syntax
- Facts and rules
- Stateless vs stateful sessions

Required Skills

- Write basic rules
- Use agenda groups
- Understand rule execution lifecycle

Assessment

- 10–15 rule exercises
- Simple eligibility ruleset
- Unit tests for rules

Certification Outcome

Certified Drools Rule Author

Level 2 – Insurance Decision Engineer (Senior)

Audience

- Senior engineers
- Claims automation engineers
- STP contributors

Competencies

- Insurance domain modeling
- Rule grouping & phasing
- Decision object patterns
- Explainability

Required Skills

- Build eligibility + routing rules
- Emit structured decision evidence
- Avoid anti-patterns (salience wars, mutation)

Assessment

- Build a Travel Delay STP ruleset
- Explain every fired rule
- Scenario-based testing

Certification Outcome

Certified Insurance Decision Engineer

Level 3 – Ruleset Architect (Staff / Principal)

Audience

- Staff engineers
- Platform architects
- Decision platform owners

Competencies

- Ruleset design
- Template-driven scaling
- Global overlay strategies
- Performance tuning

Required Skills

- Design reusable rule templates
- Support multi-region deployment
- Implement versioning strategy
- Optimize fact models

Assessment

- Design a global ruleset with regional overlays
- Implement templates + CSVs
- CI/CD promotion plan

Certification Outcome

Certified Drools Ruleset Architect

Level 4 – STP Decision Platform Architect (Expert)

Audience

- Principal engineers
- Enterprise architects
- AI + Decision leads

Competencies

- End-to-end STP architecture
- Governance & compliance
- AI + Drools coexistence
- Regulatory explainability

Required Skills

- Position Drools within FNOL → Settlement
- Design audit and replay mechanisms
- Define ownership and approval models
- Integrate AI outputs safely

Assessment

- Whiteboard full STP architecture
- Design decision governance model
- Defend design to legal/compliance

Certification Outcome

Certified STP Decision Platform Architect

Level 5 – Global Decision Authority (Distinguished)

Audience

- Global platform owners
- Architecture council members
- Decision governance leads

Competencies

- Enterprise-scale decisioning
- Regulatory strategy
- Cross-LOB reuse
- Organizational enablement

Required Skills

- Define decision standards
- Govern global rollouts
- Align AI, rules, and humans
- Mentor rule architects

Assessment

- Define enterprise decision standards
- Create certification curriculum
- Review and approve rulesets globally

Certification Outcome

Global Decision Authority – Insurance STP

Final Thought

If engineers **only know DRL syntax**, they are *rule coders*.

If they understand:

- decision boundaries
- templates
- explainability
- governance
- global deployment

They become **decision engineers**.

And at scale, **decision engineering is the real competitive advantage in insurance STP.**

Part 3

The Decision Engineering Hierarchy (Expert Model)

Think of Drools as a **runtime execution engine** inside a broader **Decision Platform**. So the hierarchy has **two layers**:

1. **Business / Platform layer** (what you govern, version, and promote)
2. **Drools / KIE layer** (what actually compiles and runs)

Layer 1: Decision Platform (Business Assets)

Decision (Capability) → Ruleset → Ruleset Version → Content Sources (Rules / Templates / Decision Tables / Params)

Layer 2: KIE Runtime (Execution Assets)

KJAR → KIE Module → KIE Base → KIE Session → Working Memory (Facts) → Agenda → Fired Rules

1) Decision (Capability) — The Missing Top Node

What it is

A **Decision** is the business capability you're automating, e.g.:

- “Travel Delay Adjudication”
- “Auto STP Routing”
- “Coverage Eligibility”

- “Payment Approval”

Why it matters

This is the unit stakeholders and auditors understand.

Relationship

A **Decision** is implemented by **one or more Rulesets** (common in large programs):

- one ruleset for eligibility
 - one for payout calculation
 - one for routing / exceptions
-

2) Ruleset (Rule Product) — The Deployable Decision Unit

What it is

A **Ruleset** is a **decision product**: a cohesive package of rules logic that answers *one business question*.

Example:

- TravelDelayAdjudicationRuleset

What it contains (conceptually)

- many rules
- grouped into phases (rule groups)
- possibly generated from templates or decision tables
- associated parameters (thresholds per region/product)
- metadata: owner, region scope, effective dates, approval state

Versioning

Rulesets MUST be versioned because they represent immutable decision logic used for audit/replay:

- TravelDelayAdjudicationRuleset v2.1.0

Relationship

- **Decision → Ruleset:** 1-to-many (sometimes 1-to-1)

- **Ruleset → Ruleset Version:** 1-to-many
- **Ruleset Version → Executable Artifact (KJAR):** 1-to-1 (recommended)

Key principle: **rules are reusable at design time**, but each *ruleset version* freezes a snapshot for runtime.

3) Ruleset Version — The Frozen Snapshot (Audit Anchor)

What it is

A **specific immutable snapshot** of a ruleset:

- exact DRL content
- exact templates/decision tables resolved
- exact parameter versions (if externalized)
- exact dependencies

Why it matters

When someone asks “*which logic was used?*” the answer is:

- ruleset name + version (+ artifact hash)

Relationship

At runtime you log:

- rulesetVersion
- which rule IDs fired
- optionally the rule content hash(es)

This is your **audit + replay** foundation.

4) Rule — The Reusable Unit of Logic (But Not a Deployable Unit)

What it is

A **Rule** is an atomic piece of logic:

- IF conditions match facts
- THEN produce decision evidence / actions

Identity vs Version

Rule ID should be stable across edits if the business intent is the same:

- POLICY_ACTIVE_ON_LOSS_DATE

Do not treat rule versioning as a deployment mechanism.

Instead:

- rule revision can exist as metadata (commit hash / revision stamp)
- but **ruleset version** is the governance version

Reuse across rulesets

This is the point you raised (correctly):

- A rule like “Policy must be active on date of loss” is used in many decisions.
- Therefore: **Rule has no intrinsic rulesetId**.
- Association is **many-to-many** via a composition record:

RulesetComposition (join)

- rulesetId
- ruleId
- ruleGroup / phase
- enabled flag
- optional overrides

Relationship

- **Rule Library → many Rulesets**
- **Ruleset Version → contains a compiled snapshot of rules**

5) Rule Group — The Execution Phase (Order + Structure)

What it is

A **Rule Group** is not a “business asset”—it’s an **execution structuring mechanism**.

Its purpose:

- make rules maintainable
- prevent salience wars

- enforce phased evaluation (validation → eligibility → calc → routing)

In Drools terms

Rule groups typically map to:

- `agenda-group` (most common)
- `ruleflow-group` (if orchestrated via BPMN / process)

Relationship

- **Ruleset contains multiple Rule Groups**
- **Rule belongs to exactly one group *within a ruleset*** (best practice)
- Same rule reused in another ruleset can be placed into a different group (often still “validation”)

Rule groups define “how to evaluate” *inside* a ruleset.
They do not define “what decision” (that’s ruleset).

6) Rule Template — Logic Pattern with Parameters (Scaling Mechanism)

What it is

A template is a **rule generator pattern**:

- stable DRL structure
- placeholders for variable content
- driven by parameter rows (CSV/DB)

Used when:

- the logic is consistent
- thresholds vary by region, product, partner, coverage, etc.

Relationship

Templates generate **many rules** that end up inside a ruleset version.

So:

- **Template** (pattern) → produces **Rules** (instances)
- **Rules** → organized into **Rule Groups**
- **Ruleset Version** → freezes generated rules

Why templates matter for global STP

They enable:

- EMEA/APAC/LATAM differences as parameters, not copy-pasted rules
 - faster rollout
 - consistent governance
-

7) Decision Table — Business-Friendly Bulk Rule Authoring

What it is

A **Decision Table** is a structured tabular way to define many rules. Drools supports decision tables (often XLS/CSV) that compile to rules.

Use when:

- business wants to manage thresholds/conditions in a tabular form
- rule logic fits a “row = rule” mental model

Avoid when:

- logic requires complex joins, temporal constraints, or deep object graph reasoning
- you need advanced control flow or multi-step reasoning

Relationship

Decision tables are another **rule source**:

- `DecisionTable` → compiled → Rules → grouped → packaged → executed

Think:

- Templates and decision tables are both *rule factories*
 - DRL is direct hand-coded rules
-

8) KIE / Drools Runtime Chain (Execution Hierarchy)

This is the “real hierarchy” in Drools runtime.

8.1 KJAR (Knowledge JAR)

A **versioned deployable artifact** containing:

- DRL
- decision tables
- templates
- rule assets
- `kmodule.xml`
- dependencies

This is what your decision service loads from artifact storage.

In enterprise STP: treat **KJAR** as the immutable runtime representation of a ruleset version.

8.2 KIE Module

The internal representation of a set of knowledge bases and sessions defined in `kmodule.xml`.

8.3 KIE Base (KieBase)

A compiled, in-memory representation of:

- rules
- processes (if any)
- type declarations

A KIE Base is relatively heavy to build; you usually:

- cache it per ruleset version
- refresh on version changes

8.4 KIE Session (KieSession / StatelessKieSession)

The actual execution context:

- Working memory (facts)
- Agenda (what can fire)
- Globals (carefully used)
- Event listeners (for audit)

For STP you typically prefer:

- `StatelessKieSession` for high volume deterministic decisions

8.5 Facts (Working Memory)

The objects inserted for evaluation:

- Claim facts
- Policy facts
- Derived facts (e.g., “PolicyIsActive”)
- Reference facts (jurisdiction configs, thresholds, etc.)

Facts are the boundary between your platform and Drools:

- keep them small, stable, and versioned as contracts

8.6 Agenda & Activations

Drools determines which rules match and puts them on the agenda.

Order can be affected by:

- agenda groups
- salience (avoid heavy use)
- ruleflow (if used)

8.7 Fired Rules → Decision Output

Rules should not mutate domain objects; they should write to a **Decision** output (your platform model):

- outcomes
- reason codes
- evidence
- routing decisions

9) Other Critical Items You’re Missing (Highly Important)

9.1 Fact Model / Canonical Decision Contract

This is the most important piece in STP:

- your facts are your “decision API contract”
- decouple facts from upstream raw payloads (FNOL schemas)
- stabilize facts to protect rules from upstream churn

9.2 Effective Dating (Temporal Validity)

Insurance decisions must support:

- effectiveStart / effectiveEnd of ruleset versions
- policy period alignment
- jurisdiction changes over time

Common pattern:

- choose ruleset version based on **loss date**, not “today”

9.3 Rule Units (Drools feature)

Rule Units can help structure rules + data sources cleanly. Not always necessary, but relevant if you want stronger modularization.

9.4 Listeners & Audit Capture

You must implement:

- rule firing listeners
- capture fired rules
- capture ruleset version
- persist decision trace

Drools don't persist this automatically.

9.5 Promotion & Governance Workflow

Your UI implies you already have this, but it must be explicit:

- Draft → Review → Approved → Promoted → Deprecated
- environment constraints (SIT/UAT/PROD)
- approval gates (architecture + business + compliance)

9.6 Artifact Registry (Ruleset Registry)

Store:

- ruleset version → artifact coordinates (groupId/artifactId/version)
- artifact hash
- effective date

- status per env

This is how runtime “knows what to load”.

10) Putting It Together — Relationships Summary

Here's the hierarchy and relationships in one view:

Business / Platform

- **Decision**
 - Ruleset
 - Ruleset Version
 - includes Rule Groups
 - contains Rules
 - rules may be hand-authored DRL or generated from
 - Templates
 - Decision Tables
 - packaged as KJAR
 - registered in Ruleset Registry
 - executed by Decision Service
 - audit persisted in Decision Trace Store

Drools / Runtime

- KJAR → KieModule → KieBase → KieSession → Facts → Agenda → Fired Rules → Decision Output
-

11) The Most Important “Insurance STP” Rule of Thumb

Reuse rules at design time via a Rule Library.
Freeze rules at runtime via Ruleset Versions (KJARs).

That gives you:

- reuse without drift
- auditability
- replay
- safe global rollout

12) Practical Guidance for Your UI + DB Model

Given your UI (“products, rules, rulesets”), the clean model is:

Tables / Entities

- Product
- Decision
- Ruleset
- RulesetVersion
- RuleLibrary (Rule)
- RulesetComposition (join)
- TemplateLibrary
- DecisionTableLibrary
- ParameterSet (optional)
- ArtifactRegistry
- ApprovalWorkflow / Status

Runtime resolution

- Product + Decision + Loss Date + Region → choose RulesetVersion → load KJAR → run Drools

Part 4

Rulesets vs Rule Groups — The Mental Model That Finally Clicks (Executive Summary)

Concept	What it is	Why it exists
Ruleset	A business decision product	Defines <i>what decision is being made</i>
Rule Group	An execution phase inside a ruleset	Defines <i>how the decision is evaluated</i>

Rulesets are about ownership and boundaries.
Rule groups are about execution order and structure.

They solve different problems.

Start With the Business Question (Not Drools)

Example Business Question

“Is this Travel Delay claim eligible for automatic payout, and if so, how much?”

That question is **one decision**.

That decision → **ONE RULESET**

Your intuition is correct here.

What Is a Ruleset (Proper Definition)

A **ruleset** is:

A versioned, deployable, independently testable decision artifact that owns **one business decision**.

Example Ruleset

`TravelDelayAdjudicationRuleset`

It owns:

- Eligibility
- Threshold checks
- Regulatory constraints
- Payout calculation
- Final routing decision

It has:

- A version (v1.3.0)
- Owners (Claims + Platform)
- A contract (input facts → decision output)

Think of a ruleset like a **microservice**, but for decisions.

Then Why Do Rule Groups Exist?

Because **one decision is never evaluated in one flat “if-then” block**.

Inside that Travel Delay decision, you still need **structure**.

What Is a Rule Group (Proper Definition)

A **rule group** is:

A **logical execution phase inside a ruleset** that organizes *how* rules are evaluated.

Rule groups exist to:

- Avoid salience wars
- Make rule execution understandable
- Enable phased decisioning
- Improve testability and explainability

They do **NOT**:

- Define business ownership
 - Get deployed independently
 - Represent a business decision
-

Concrete Example (This Is the Key)

Ruleset

TravelDelayAdjudicationRuleset

Inside That Ruleset

```
rules/
  └── 01_validation/
  └── 02_normalization/
  └── 03_eligibility/
  └── 04_calculation/
  └── 05_exclusions/
  └── 06_routing/
  └── 07_final_decision/
```

Each folder maps to a **rule group**.

Why This Separation Matters (This Is the “Aha”)

If You Only Think in Rulesets

You get:

- Huge DRL files
- Hard-to-follow logic
- Fragile salience
- Poor explainability

If You Only Think in Rule Groups

You get:

- Technical organization
- But no business boundaries
- Rules reused incorrectly
- Governance chaos

When You Separate Them Correctly

You get:

- Clear decision ownership (**ruleset**)

- Clean execution flow (**rule groups**)
 - Reusable patterns
 - Scalable governance
-

Analogy (This Usually Makes It Click)

Ruleset = Microservice

- Has a purpose
- Has an API
- Has owners
- Is versioned and deployed

Rule Groups = Methods inside the Service

- Validation
- Processing
- Calculation
- Finalization

You wouldn't deploy a method by itself.

You wouldn't mix two microservices into one.

Same idea.

How This Applies to Your Example (Travel Delay)

Your mental model:

“Rulesets are a grouping of rules for travel delay adjudication”

 **Correct — but incomplete**

Corrected expert model:

“A ruleset is the deployable decision product for travel delay adjudication, and rule groups are the internal execution phases that structure how that decision is evaluated.”

Another Example (To Cement It)

Separate Rulesets (Different Decisions)

Ruleset	Decision
TravelDelayEligibilityRuleset	Is claim eligible?
TravelDelayPayoutRuleset	How much to pay?
TravelDelayRoutingRuleset	STP or manual?

Each could be:

- Independently versioned
- Independently reused
- Independently tested

Each Ruleset Still Has Rule Groups

```
Eligibility Ruleset
└── validation
└── eligibility
└── final_decision
```

What About Templates?

Templates live at:

- **Rule group level** (logic pattern)
- **Ruleset scope** (business context)

They **do not define a ruleset**, they **populate rule groups**.

Common Mistake (Very Important)

- ✗ Treating rule groups as reusable business assets
- ✗ Treating rulesets as just folders

This leads to:

- Hidden coupling
- Unclear ownership
- Regulatory risk

The One-Sentence Distinction (Memorize This)

Rulesets define *what decision* is being made.

Rule groups define *how that decision is evaluated*.

If you remember only one thing, remember that.

Final Check (If This Is True, You've Got It)

Ask yourself:

- Can I version this independently? → **Ruleset**
 - Is this a phase of evaluation? → **Rule Group**
 - Does the business care about this artifact? → **Ruleset**
 - Does engineering care about execution order? → **Rule Group**
-

If you want next, I can:

- Walk through **one Travel Delay ruleset end-to-end** with real rule groups
- Show **bad vs good** designs side-by-side
- Map this directly to **Drools constructs** (agenda-group, ruleflow-group)
- Tie this into **CDF + FNOL Evaluator**

Just say where you want to go next.

So, a rule product, ruleset, contains multiple rules. A rule group defines the order, grouping of execution of those rules

Can you answer the question, comment above?

Yes — **your statement is correct**, and I'll refine it into the **precise, expert-level version** so it's unambiguous and future-proof.

Short, Correct Answer (Yes, with Precision)

A ruleset (rule product) contains multiple rules that together implement one business decision.

A rule group defines how subsets of those rules are grouped and executed in a controlled order within that ruleset.

That statement is **100% correct**.

Now let me tighten the language so it's architecturally precise.

Corrected Expert Definition (What You Should Internalize)

Ruleset (Rule Product)

- A **deployable, versioned decision unit**
- Owns **one business decision**
- Contains:
 - Multiple rules
 - Multiple rule groups
 - Templates and parameters
- Has:
 - Business ownership
 - A decision contract
 - Independent lifecycle

Think:

“This ruleset answers a specific insurance question.”

Rule Group

- A **logical execution phase** inside a ruleset
- Groups rules by **when and why** they execute
- Controls:
 - Order of execution
 - Scope of evaluation
 - Decision flow clarity
- Exists **only inside** a ruleset

Think:

“This rule group represents a step in evaluating the decision.”

One-Sentence Mental Model (Perfect for Teaching Others)

Rulesets define *what decision* is being made.

Rule groups define *how that decision is evaluated*.

Concrete Example (Travel Delay)

Ruleset

TravelDelayAdjudicationRuleset

Rule Groups Inside It

validation
eligibility
calculation
exclusions
routing
final_decision

Each group:

- Contains multiple rules
 - Executes in a defined sequence
 - Contributes evidence to the final decision
-

What Rule Groups Are NOT

- ✗ Not independently deployable
- ✗ Not business-owned assets
- ✗ Not reusable across unrelated decisions

They are **structural**, not **product-level**.

Why This Matters in STP

If you blur these concepts:

- Governance breaks

- Explainability suffers
- Versioning becomes dangerous
- Global rollout becomes brittle

When you keep them distinct:

- Decisions scale cleanly
 - Teams align faster
 - Compliance becomes tractable
-

Final Validation (Your Original Statement, Refined)

Your original statement:

“A rule product, ruleset, contains multiple rules. A rule group defines the order, grouping of execution of those rules.”

Expert version:

A ruleset is a decision product that contains multiple rules.

Rule groups organize those rules into execution phases that control how the decision is evaluated.

You've got it.

Are rulesets versioned or only the rules inside it?

This is a **very important question**, because the answer determines whether your platform is **safe, auditable, and scalable** or **fragile and dangerous**.

I'll give you the **short correct answer first**, then the **architectural reasoning**, then **what actually works in enterprise insurance**.

Short, Correct Answer

Rulesets are versioned.

Individual rules inside a ruleset are *not* versioned independently in production.

That is the **correct enterprise answer**.

Why This Is the Correct Model (Architect View)

What the Business Cares About

The business, regulators, and auditors care about:

- “*Which decision logic version was used?*”
- “*Which eligibility criteria applied on this claim?*”

They do **not** care that:

- Rule `ELIGIBILITY_17` changed salience
- Rule `CALC_05` was edited internally

They care about the **decision as a whole**.

That decision = **Ruleset**.

What a Ruleset Version Represents

A ruleset version represents:

The complete, immutable decision logic as executed at a point in time.

It includes:

- All rules
- Their conditions and consequences
- Rule groups and execution order
- Templates + resolved parameters
- Any decision metadata

So when you say:

`TravelDelayAdjudicationRuleset v1.3.0`

You are saying:

“This exact decision logic was applied.”

Why You Should NOT Version Individual Rules Independently

1. Rules Are Not Executed in Isolation

- Rules interact
- Ordering matters
- Side effects matter (decision evidence)

Changing one rule changes the behavior of the entire decision.

2. Independent Rule Versioning Breaks Auditability

Imagine telling a regulator:

“We used Rule A v2.1, Rule B v1.7, Rule C v3.4...”

That is **not defensible**.

Auditors want:

“We used *this decision logic*.”

3. Deployment Becomes Dangerous

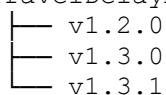
If rules are versioned independently:

- Partial upgrades occur
 - Inconsistent decisions happen
 - Rollbacks become impossible
-

Correct Enterprise Versioning Model

Version at the Ruleset Level

TravelDelayAdjudicationRuleset



Each version is:

- Immutable
 - Fully testable
 - Fully auditible
-

What About Templates and Parameters?

This is where nuance comes in.

Templates

- Versioned with the ruleset
- Changes → new ruleset version

Parameters (CSV / Config)

Two models:

Model A (Safer, Preferred)

- Parameters are **packaged into the ruleset**
- Any change → new ruleset version

Model B (More Flexible, Riskier)

- Parameters versioned separately
- Ruleset references parameter version
- Decision audit must record both versions

In insurance STP, **Model A is strongly preferred**.

Can Rules Have IDs or Metadata?

Yes — **but that is not versioning**.

Inside rules you **SHOULD** have:

```
@ruleId("TD_ELIGIBILITY_001")
@ruleGroup("eligibility")
@severity(HIGH)
```

This helps with:

- Explainability
- Audit logs
- Debugging

But the **version still belongs to the ruleset**, not the rule.

Analogy (This Makes It Crystal Clear)

Software World

- You version the **service**
- You don't version each `if` statement

Drools World

- You version the **ruleset**
- You don't version each rule independently

Same principle.

What Happens When a Single Rule Changes?

Even if:

- One threshold changes
- One condition is updated
- One exception is added

👉 You release a new ruleset version.

Example:

- v1.3.0 – logic change
 - v1.3.1 – threshold update
 - v1.4.0 – new eligibility criteria
-

Final Rule (Memorize This)

If a change can affect a decision outcome, it requires a new ruleset version.

That rule will keep you safe.

One-Line Answer (If Someone Asks You in a Meeting)

"We version rulesets, not individual rules. A ruleset version represents the complete decision logic used for audit and replay."

If you want next, I can:

- Show **semantic versioning rules for insurance**
- Explain **how to hotfix thresholds safely**
- Map versioning to **CI/CD and promotion**
- Show **how CDF and Drools coexist with versioning**

Just tell me.

but should i be able to say 'in this ruleset i used version 2 of rule XYZ?

Short answer: **No — not in the way you're thinking.**

Long answer (the important one): **you can reference rule identity and revision for traceability, but you must not treat individual rules as independently versioned decision assets.**

Let me be very precise, because this is where a lot of teams accidentally design themselves into audit hell.

The Correct, Enterprise-Safe Answer

You should never say:

"This ruleset uses version 2 of rule XYZ"
as a *decision or deployment concept*.

You *may* say it **only as internal traceability metadata**, never as a governance or release construct.

Why This Distinction Matters

There are **two very different notions** that often get conflated:

Concept	Allowed?	Purpose
Ruleset versioning	<input checked="" type="checkbox"/> Yes	Decision governance, audit, replay
Rule identity / revision metadata	<input checked="" type="checkbox"/> Yes	Traceability, debugging
Independent rule versioning	<input type="checkbox"/> No	Breaks audit & determinism

The Key Principle (Memorize This)

Rules are not decision units.

Rulesets are decision units.

Once a rule participates in a ruleset, it **loses independent semantic meaning** from a governance standpoint.

What You *Should* Say Instead (Correct Language)

Incorrect (Dangerous)

“This ruleset uses version 2 of rule XYZ.”

This implies:

- Rules are independently versioned
- Rules can be swapped in/out safely
- Partial upgrades are acceptable

All of that is false in regulated insurance STP.

Correct (Enterprise-Safe)

“This decision was produced by `TravelDelayAdjudicationRuleset v2.1.0`, and rule `TD_ELIGIBILITY_MAX_DELAY` fired with revision hash `a7f9c2`.”

Notice the difference:

- **Versioning is at the ruleset**
 - **Rule detail is traceability, not governance**
-

How to Model This Correctly in Practice

1. Rules Have Stable IDs (Not Versions)

Each rule should have:

- A stable identifier
- Optional internal revision metadata

Example:

```
rule "Travel Delay - Max Delay Threshold"
@ruleId("TD_ELIGIBILITY_MAX_DELAY")
@ruleGroup("eligibility")
@revision("2024-11-15")
when
    ...
then
    ...
end
```

This allows you to say:

- “This rule fired”
- “This rule was last modified on X”

But **not**:

- “We deployed rule version 2 independently”
-

2. Ruleset Owns the Version

```
TravelDelayAdjudicationRuleset
├── v2.0.0
└── v2.1.0
    └── v2.1.1
```

Each version represents:

- A frozen snapshot of **all rules**
- Their interactions

- Their execution order
 - Their parameters
-

3. Audit Records Look Like This (Correct)

```
{  
  "ruleset": "TravelDelayAdjudicationRuleset",  
  "rulesetVersion": "2.1.0",  
  "firedRules": [  
    {  
      "ruleId": "TD_ELIGIBILITY_MAX_DELAY",  
      "ruleGroup": "eligibility",  
      "revision": "2024-11-15"  
    }  
  ]  
}
```

This is **perfect** for:

- Regulators
 - Legal
 - Claim disputes
 - Replay
-

Why Independent Rule Versioning Is a Trap

Let's say you *did* allow this:

“Ruleset A uses rule X v2 and rule Y v3”

Now answer these questions (you can't, safely):

- What happens if rule X v2 assumes rule Y v2 behavior?
- What if rule ordering changed between versions?
- How do you replay a historical decision?
- How do you roll back *only one rule* safely?

This leads to:

- Hidden coupling
 - Non-deterministic outcomes
 - Regulatory exposure
-

The One Exception (Very Narrow, Advanced)

There *is* one advanced pattern where rule “versions” appear — **templates**.

But even there:

- The resolved rules are **compiled into the ruleset**
- The ruleset version still owns the decision

You may say internally:

“This ruleset was generated from template v3”

But externally and legally:

“Ruleset v2.1.0 was used”

The Gold-Standard Answer (If Asked by an Architect or Auditor)

“We do not version individual rules as decision assets.

We version rulesets.

Individual rules have stable IDs and revision metadata purely for traceability within a ruleset version.”

That answer will **always** pass architecture, audit, and compliance review.

Final Mental Model (Lock This In)

- **Ruleset version** → *What decision logic was used*
- **Rule ID** → *Which logic fired*
- **Rule revision** → *Internal traceability*
- **Never** → independent rule versions in production

If you want next, I can:

- Show **how to implement this cleanly in Drools**
- Map this to **Git commits vs releases**
- Show **how to hotfix a single threshold safely**
- Tie this directly into **CDF + AI-generated rules**

Just say where to go next.

So, if I defined rule 1 and it's used in ruleset 'traveldelay' and then this rule is changed, how do I know which version of rule 1 was used?

Great question — this is **the exact edge case** where people either design a **clean, auditable decision platform**... or accidentally create something they can't defend later.

Let's answer it **precisely**, step by step, without hand-waving.

Short, Correct Answer

You do NOT track “which version of rule 1 was used.”

You track:

- 1) Which *ruleset version* was used
- 2) Which *rule ID* fired
- 3) The *immutable rule content* as it existed in that ruleset version

That combination tells you **exactly** which version of the rule logic was used — without independently versioning the rule.

The Key Idea (This Is the Mental Shift)

Rules do not have versions at runtime.

Rulesets freeze rule content.

When a rule changes:

- The **ruleset version changes**
- The **old rule logic still exists**, frozen inside the old ruleset version

So you never ask:

“Which version of rule 1 was used?”

You ask:

“Which ruleset version was used, and which rule ID fired?”

That is enough to reconstruct the exact logic.

Walk Through Your Exact Scenario

Step 1: You Define Rule 1

```
rule "Max Delay Threshold"
@ruleId("TD_RULE_001")
when
    Claim(delayMinutes > 120)
then
    decision.reject("DELAY_TOO_SHORT");
end
```

This rule is packaged into:

TravelDelayRuleset v1.0.0

Step 2: A Claim Is Processed

Audit record:

```
{
  "ruleset": "TravelDelayRuleset",
  "rulesetVersion": "1.0.0",
  "firedRules": ["TD_RULE_001"]
}
```

At this point:

- The **exact DRL content** of rule TD_RULE_001
 - Is frozen inside **ruleset v1.0.0**
-

Step 3: You Change Rule 1

You update the rule:

```
rule "Max Delay Threshold"
@ruleId("TD_RULE_001")
when
    Claim(delayMinutes > 90)
then
    decision.reject("DELAY_TOO_SHORT");
end
```

⚠ This is a **behavior change**, so:

TravelDelayRuleset v1.1.0

is created.

Step 4: Another Claim Is Processed

Audit record:

```
{  
  "ruleset": "TravelDelayRuleset",  
  "rulesetVersion": "1.1.0",  
  "firedRules": ["TD_RULE_001"]  
}
```

Now Answer the Question Directly

How do I know which version of rule 1 was used?

You know it because:

1. **Ruleset version is recorded** (v1.0.0 vs v1.1.0)
2. Each ruleset version contains an **immutable snapshot of rule 1**
3. Rule ID tells you *which rule fired*
4. Ruleset version tells you *which exact logic that rule had*

There is **no ambiguity**.

How This Looks in a Real Audit or Dispute

An auditor asks:

“Which rule logic rejected this claim on March 12?”

You answer:

“The claim was processed using TravelDelayRuleset v1.0.0.
Rule TD_RULE_001 fired.
Here is the exact DRL content of that ruleset version.”

That is:

- Clear
 - Defensible
 - Replayable
 - Legally sound
-

Optional (But Strongly Recommended): Rule Content Hashing

For extra rigor, many enterprise platforms also record a **rule hash**.

At build time:

```
TD_RULE_001 hash = a9f3c2e
```

Audit record:

```
{
  "ruleset": "TravelDelayRuleset",
  "rulesetVersion": "1.0.0",
  "firedRules": [
    {
      "ruleId": "TD_RULE_001",
      "ruleHash": "a9f3c2e"
    }
  ]
}
```

This gives you **cryptographic certainty**.

What You Should NOT Do (Very Important)

✗ Do NOT say:

“We used rule 1 version 2”

That implies:

- Rules are deployed independently
- Partial upgrades are possible
- Decision logic is composable at runtime

All of that is unsafe in insurance STP.

The One-Sentence Answer You Can Use Anywhere

“We determine which rule logic was used by recording the ruleset version and the rule ID that fired. Each ruleset version freezes the exact rule content.”

If you remember that sentence, you’re safe.

Final Mental Model (Lock This In)

- **Ruleset version** = frozen decision logic
 - **Rule ID** = traceability
 - **Rule content** = immutable within that ruleset
 - **No independent rule versions in production**
-

If you want, next I can:

- Show **how to implement this cleanly in Drools (code + build)**
- Map this to **Git commits vs release tags**
- Show **how to replay historical decisions**
- Tie this into **CDF / AI-generated rules governance**

Just tell me where to go next.

ok, if i have rule1 was id 10 and then i modify rule1 then in my persistence layer a new entry for rule1 is created with a different id?

Short answer: **No — you should NOT create a new rule ID in your persistence layer when you modify the rule.**

Long answer (the one that will keep you safe in audits and at scale):

The Correct, Enterprise-Safe Model

Rule ID is stable.

Rule content changes.

Ruleset version changes.

You do **not** create a new rule ID when a rule is modified **if the rule still represents the same business intent.**

How to Think About Rule IDs (This Is the Key)

Rule ID answers:

“What business rule is this?”

Ruleset version answers:

“Which exact logic was used?”

These are **orthogonal concerns**.

Your Example, Walked Through Correctly

Step 1: Original Rule

```
rule "Max Delay Threshold"
@ruleId("10")
when
    Claim(delayMinutes > 120)
then
    decision.reject("DELAY_TOO_SHORT");
end
```

This rule lives inside:

TravelDelayRuleset v1.0.0

Persistence (conceptually):

```
{
  "ruleId": "10",
  "businessMeaning": "Max travel delay threshold",
  "rulesetVersion": "1.0.0"
}
```

Step 2: You Modify the Rule Logic

Change:

```
delayMinutes > 120
```

to:

```
delayMinutes > 90
```

What happens?

Ruleset version increments

TravelDelayRuleset v1.1.0

Rule ID does NOT change

Why?

- It's still the *same business rule*
 - Only the threshold changed
-

Step 3: How Persistence Should Look

You now have:

```
Rule ID: 10
└─ Appears in ruleset v1.0.0 (delay > 120)
   └─ Appears in ruleset v1.1.0 (delay > 90)
```

Each ruleset version contains a **frozen copy of the rule content**.

What Actually Gets Persisted (Correct Model)

In production-grade platforms, you persist:

1. Ruleset Version (Primary Key)

```
{
  "ruleset": "TravelDelayRuleset",
  "version": "1.1.0",
  "artifactHash": "abc123"
}
```

2. Fired Rule Metadata (Per Decision)

```
{  
  "ruleset": "TravelDelayRuleset",  
  "rulesetVersion": "1.1.0",  
  "firedRules": [  
    {  
      "ruleId": "10",  
      "ruleHash": "f92e7a"  
    }  
  ]  
}
```

You do **not** need a separate “rule table” with versions for decision replay.

When *Would* a New Rule ID Be Created?

Only when **business meaning changes**, not logic tuning.

New Rule ID

- “Max delay threshold” → “Minimum delay for compensation”
- Eligibility rule → Fraud indicator
- Coverage exclusion → Regulatory exception

In other words:

If a business SME would say “that’s a different rule,” it gets a new ID.

Why Creating a New Rule ID on Every Change Is a Bad Idea

If you did that:

- ✗ You lose semantic continuity
- ✗ Rule analytics become meaningless
- ✗ Audits become noisy
- ✗ SMEs can’t track rule behavior over time

You’d end up with:

```
Rule 10  
Rule 10_v2  
Rule 10_v3
```

That's **source control thinking**, not **decision governance thinking**.

The Gold-Standard Pattern (What Mature Platforms Do)

Concept	Stable?	Versioned?
Rule ID	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No
Rule Content	<input type="checkbox"/> No	Implicit via ruleset
Ruleset	<input type="checkbox"/> No	<input checked="" type="checkbox"/> Yes
Decision Output	<input type="checkbox"/> No	Immutable

One-Sentence Answer You Can Use Anywhere

No — modifying a rule does not create a new rule ID.

The rule ID stays stable, and the ruleset version changes, which freezes the rule's content.

If you want next, I can:

- Show a **concrete DB schema** for this
- Map this to **Git commits vs releases**
- Show **how to do hotfixes safely**
- Tie this into **AI-generated rules governance**

Just say the word.

is this something handled internally by drools or the developer needs to take care of it?

This is the **right final question** in the sequence — because it draws the **hard boundary** between what Drools gives you and what *you*, as a platform architect, must own.

Short, Correct Answer

Drools does NOT handle this for you.

The developer / platform must take care of it.

Drools is **deliberately opinionated only about rule execution**, not about:

- versioning
- governance
- audit
- persistence
- compliance

Those are **your responsibility**.

What Drools Handles vs What It Does NOT

✓ What Drools Handles Internally

Drools takes care of:

- Rule matching (Rete algorithm)
- Rule firing
- Agenda / ruleflow execution
- Fact lifecycle
- Conflict resolution
- Stateless vs stateful execution
- Rule metadata attached to a rule instance

In short:

Drools answers: “Given this set of rules and facts, what fires?”

✗ What Drools Does NOT Handle (By Design)

Drools does **not** handle:

- Rule versioning
- Ruleset versioning
- Decision lifecycle
- Rule identity semantics
- Audit persistence
- Replayability
- Regulatory traceability
- Environment promotion
- Governance approvals

Drools intentionally avoids this because:

- It's domain-specific
 - It varies by industry
 - It's platform responsibility
-

Why Drools Does NOT Version Rules

Drools treats rules as:

- **Compiled artifacts**, not governed assets
- Loaded at runtime
- Executed in-memory

Once a KIE base is built:

- Drools has **no concept of “previous versions”**
- No built-in persistence of rule history
- No audit trail beyond execution callbacks

This is **by design**, not a limitation.

So Who Owns This? (The Real Answer)

You do — at the Decision Platform layer

In an insurance STP platform, this responsibility typically sits in:

- A **Decision Service**
 - A **Ruleset Registry**
 - A **Decision Governance Layer**
 - Or what you might call **CDF / Decision Orchestrator**
-

The Correct Responsibility Split (Very Important)

Concern	Owner
Rule execution	Drools
Rule grouping	Drools
Ruleflow	Drools

Concern	Owner
Ruleset versioning	Platform
Rule identity	Platform
Decision audit	Platform
Replay	Platform
Promotion	Platform
Compliance	Platform

This split is **intentional and healthy**.

What You Must Implement (Minimum Set)

To do this correctly, your platform must implement:

1. Ruleset Packaging

- Each ruleset version is built as an immutable artifact (KJAR or equivalent)
- Contains all rules, templates, parameters

2. Ruleset Version Registry

- Name
- Version
- Hash
- Environment
- Status (DEV / UAT / PROD)

3. Execution Context

At runtime, inject:

- Ruleset name
- Ruleset version
- Correlation ID
- Environment

4. Decision Audit Persistence

Persist:

- Ruleset name + version
- Fired rule IDs

- Rule hashes (optional but recommended)
 - Decision output
-

How This Looks in Practice (Conceptual)

```
[Decision API]  
↓  
Load Ruleset v2.1.0  
↓  
Execute Drools  
↓  
Capture fired rules  
↓  
Persist audit record
```

Drools:

- Executes rules

Your platform:

- Knows **which artifact was loaded**
 - Knows **why it was loaded**
 - Knows **what fired**
 - Knows **how to replay it later**
-

Why This Is Actually a Good Thing

If Drools tried to do this internally:

- It would force one governance model
- It would break flexibility
- It would be wrong for many domains

Insurance STP requires:

- Stronger audit
- Longer retention
- Regulatory defensibility

That belongs **outside the engine**.

One-Sentence Answer You Can Use Anywhere

Drools executes rules, but it does not manage rule or ruleset versioning, audit, or governance — that is the responsibility of the decision platform built around Drools.

That sentence is **100% accurate** and will stand up in architecture review.

Final Mental Model (Lock This In)

- Drools = **CPU**
 - Your platform = **Operating System**
 - Governance, versioning, audit = **Your job**
-

If you want next, I can:

- Design a **Ruleset Registry schema**
- Show a **reference Decision Service architecture**
- Provide **sample code (Java / .NET)**
- Map this cleanly to **CDF + AI-generated rules**
- Show how **Foundry / AI Gateway fits safely**

Just tell me where you want to go next.

We have a UI where we specify for a given business product rules and rulesets. The database that i use to persist that data is it a drool database or a custom one. If custom how does drools know how to load things are runtime

This is a **core architecture question**, and the answer is decisive for whether your platform scales cleanly or becomes tightly coupled to Drools internals.

I'll answer it **cle

dont see your answer

Got it — thanks for the nudge. I'll answer **cleanly, completely, and unambiguously**.

Short, Correct Answer (No Ambiguity)

You use a custom database, not a “Drools database.”
Drools does not persist or load rulesets for you.
Your platform loads rules into Drools at runtime.

Everything after that sentence is **platform architecture**, not Drools magic.

There Is No Such Thing as a “Drools Database”

Drools provides:

- A **rules engine**
- A **compiler**
- A **runtime (KIE)**

Drools does **not** provide:

- A database schema
- A persistence model for rules
- A product / ruleset catalog
- Version management
- Governance metadata
- UI integration

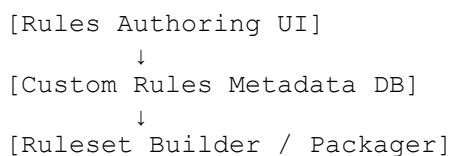
So when you build a UI to manage:

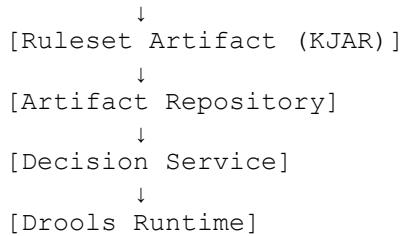
- Business products
- Rulesets
- Rules
- Parameters
- Versions

👉 That data lives in your own custom persistence layer.

This is **by design**, not a limitation.

The Correct High-Level Architecture





Drools only appears **at the very end**.

What Lives in Your Custom Database

Your UI-driven database typically stores **metadata**, not executable rules.

Example Tables / Collections

1. Product

```
{
  "productId": "TRAVEL",
  "name": "Travel Insurance"
}
```

2. Ruleset

```
{
  "rulesetId": "TRAVEL_DELAY",
  "productId": "TRAVEL",
  "status": "APPROVED"
}
```

3. Ruleset Version

```
{
  "rulesetId": "TRAVEL_DELAY",
  "version": "2.1.0",
  "artifactId": "traveldelay-2.1.0.jar",
  "status": "PROD"
}
```

4. Rule Metadata (Not Logic)

```
{
  "ruleId": "TD_RULE_010",
  "businessMeaning": "Max delay threshold",
  "rulesetId": "TRAVEL_DELAY"
}
```

⚠ You are NOT storing DRL for execution here
You are storing **business metadata and governance state**.

So Where Does the Actual Rule Logic Live?

In a Ruleset Artifact (KJAR)

At **build time**, your platform:

1. Reads approved rule definitions + templates
2. Generates DRL
3. Packages everything into a **KJAR**
4. Publishes it to an artifact repository

This artifact is:

- Immutable
 - Versioned
 - Auditable
 - Deployable
-

How Drools Loads Rules at Runtime (This Is the Key)

Drools does **not** query your database.

Instead:

Step-by-Step Runtime Flow

1. **Decision Service receives a request**
2. It determines:
 - Product
 - Ruleset
 - Version (or “latest approved”)
3. It loads the **corresponding artifact**
4. It initializes a **KIE Container**
5. It executes Drools
6. It captures fired rules
7. It persists audit output

Example (Conceptual Java)

```
KieContainer container =  
    kieServices.newKieContainer(releaseId);  
  
KieSession session =  
    container.newStatelessKieSession();  
  
session.execute(facts);
```

Drools only sees:

- Compiled rules
- Facts

It has **no idea** about your UI, DB, or products.

Why This Separation Is Critical

If Drools loaded rules directly from your DB:

- You couldn't version safely
- You couldn't promote environments
- You couldn't replay decisions
- You'd violate audit expectations

Instead:

- **DB = governance & intent**
 - **Artifact = executable truth**
-

Who Decides Which Ruleset to Load?

👉 Your Decision Orchestrator / API

This component:

- Maps business product → ruleset
- Applies versioning rules
- Enforces environment constraints
- Injects execution context

Drools is completely agnostic.

Common (Incorrect) Mental Model

 “Drools reads rules from the DB”

This leads to:

- Dynamic runtime changes
 - Non-repeatable decisions
 - Audit nightmares
-

Correct Mental Model (Use This)

 “Drools executes a pre-built, immutable ruleset artifact that my platform selected and loaded.”

That is the **enterprise-safe model**.

One-Sentence Answer You Can Use in Architecture Review

“We persist product, ruleset, and governance metadata in a custom database. At build time we generate versioned ruleset artifacts, and at runtime our decision service loads the appropriate artifact into Drools.”

That sentence is **exactly right**.

Final Lock-In Mental Model

- UI → **metadata**
 - DB → **governance**
 - Artifact → **truth**
 - Drools → **execution only**
-

If you want next, I can:

- Design a **full DB schema**
- Show a **ruleset builder pipeline**
- Provide **runtime loading code**
- Map this to AKS / CI/CD
- Tie this into CDF + AI-generated rules

Just tell me where to go next.

So, since rules can be reused across business products, for instance, policy has to be valid for us to pay claim, why in your example an attribute of a rule is the ruleset id. What happens if, in like my example, the same rule is associated with multiple rulesets

Excellent catch — you just hit a **subtle but critical architectural point**, and your instinct is **100% right**.

You are exposing the exact place where **rule reuse**, **ruleset ownership**, and **governance** must be handled *very deliberately*.

Let me correct the earlier simplification and give you the **proper enterprise model**.

The Core Correction (Very Important)

A rule must NOT be owned by a single ruleset.

Rules can be reused across multiple rulesets.

Therefore, a rule should NOT have `rulesetId` as an intrinsic attribute.

That earlier example used `rulesetId` only as a **didactic shortcut**, not as a correct final design.

Your example — “*policy must be valid to pay claim*” — is exactly the right one.

The Correct Mental Model (This Is the Key)

You need to separate **three different concerns**:

Concept	What it represents
Rule	A reusable unit of business logic (intent)
Ruleset	A decision product (business decision)
Ruleset Composition	Which rules are included in which ruleset

This implies a **many-to-many relationship**.

Correct Model: Rule Library + Ruleset Composition

1. Rule (Reusable Asset)

A **rule** represents **one business intent**, reusable across products and decisions.

Example:

“Policy must be active on date of loss”

Rule metadata (correct)

```
{  
  "ruleId": "POLICY_ACTIVE_ON_LOSS_DATE",  
  "businessMeaning": "Policy must be active on loss date",  
  "domain": "Policy",  
  "severity": "BLOCKING",  
  "owner": "Underwriting/Claims"  
}
```

 Notice:

- **No rulesetId**
 - Rule is **globally reusable**
 - Stable identity
-

2. Ruleset (Decision Product)

A **ruleset** owns a **business decision**, not the rules themselves.

Examples:

- TravelDelayAdjudicationRuleset
- AutoCollisionSettlementRuleset
- HealthClaimEligibilityRuleset

```
{  
  "rulesetId": "TRAVEL_DELAY_ADJ",
```

```
"decision": "Determine travel delay payout eligibility"  
}
```

3. Ruleset Composition (The Missing Piece)

This is where the association happens.

RulesetRule (join entity)

```
{  
  "rulesetId": "TRAVEL_DELAY_ADJ",  
  "ruleId": "POLICY_ACTIVE_ON_LOSS_DATE",  
  "ruleGroup": "validation",  
  "enabled": true  
}
```

And for another ruleset:

```
{  
  "rulesetId": "AUTO_COLLISION_ADJ",  
  "ruleId": "POLICY_ACTIVE_ON_LOSS_DATE",  
  "ruleGroup": "validation",  
  "enabled": true  
}
```

- Same rule
- Multiple rulesets
- Different decisions
- Same business meaning

This is the **correct enterprise design**.

Why Ruleset Does NOT Own the Rule

Because rules like:

- Policy active
- Coverage exists
- Claim date within policy term

Are **cross-cutting invariants**, not decision-specific logic.

If you embedded them inside one ruleset:

- You'd duplicate logic
 - You'd drift behavior
 - You'd lose governance clarity
-

How This Works at Build Time (Very Important)

At **build/package time**, your platform:

1. Takes a ruleset definition
2. Resolves all associated rules (from the rule library)
3. Generates DRL
4. Freezes them into a **ruleset version**

So at runtime:

- The rules are **no longer shared**
- Each ruleset version has its own **immutable snapshot**

This is how you get:

- Reuse at design time
 - Isolation at runtime
-

What Happens When a Shared Rule Changes?

Let's walk through your exact concern.

Rule changes:

Policy active logic changes

What happens?

- The **rule definition changes**
- Existing ruleset versions are NOT touched

Then:

- Any ruleset that wants the change
- Must release a **new ruleset version**

So you might get:

```
TravelDelayAdjudicationRuleset v2.1.0 (old rule)
AutoCollisionAdjudicationRuleset v3.4.0 (new rule)
```

This is:

- Safe
 - Explicit
 - Auditable
-

Why This Is the Only Safe Model in Insurance

If rule changes automatically propagated to all rulesets:

- You would silently change decisions
- You would break regulatory traceability
- You could not replay historical claims

This is **not acceptable** in insurance.

One-Sentence Correct Answer to Your Question

A rule does not belong to a single ruleset.

Rules live in a shared rule library, and rulesets reference them via composition. At build time, rules are copied into immutable ruleset versions.

That sentence is *architecturally correct*.

Final Mental Model (Lock This In)

- **Rule** = reusable intent
 - **Ruleset** = decision product
 - **Ruleset version** = frozen executable truth
 - **Association** = many-to-many
 - **Reuse happens at design time, isolation at runtime**
-

If you want next, I can:

- Draw the **ER model** for this
- Show **build-time resolution logic**
- Show **how Drools templates enable this**
- Map this cleanly to **AI-assisted rule authoring**
- Show **anti-patterns to avoid** (very useful)

Just say the word.

Go deep into ruleset, rule, rule group, rule template, decision table, kie, and any other item of importance i am missing in that chain. Talk about their hierarchy and relationships. Please go expert level

Can you answer my request

Absolutely. Here's the **expert-level, platform-architect view** of the full chain—**ruleset, rule, rule group, rule template, decision table, KIE**, plus the other critical artifacts you're missing—with **hierarchy + relationships** and how this all fits for **insurance STP at Chubb scale**.

The Decision Engineering Hierarchy (Expert Model)

Think of Drools as a **runtime execution engine** inside a broader **Decision Platform**. So the hierarchy has **two layers**:

1. **Business / Platform layer** (what you govern, version, and promote)
2. **Drools / KIE layer** (what actually compiles and runs)

Layer 1: Decision Platform (Business Assets)

Decision (Capability) → Ruleset → Ruleset Version → Content Sources (Rules / Templates / Decision Tables / Params)

Layer 2: KIE Runtime (Execution Assets)

KJAR → KIE Module → KIE Base → KIE Session → Working Memory (Facts) → Agenda → Fired Rules

1) Decision (Capability) — The Missing Top Node

What it is

A **Decision** is the business capability you're automating, e.g.:

- “Travel Delay Adjudication”
- “Auto STP Routing”
- “Coverage Eligibility”
- “Payment Approval”

Why it matters

This is the unit stakeholders and auditors understand.

Relationship

A **Decision** is implemented by **one or more Rulesets** (common in large programs):

- one ruleset for eligibility
 - one for payout calculation
 - one for routing / exceptions
-

2) Ruleset (Rule Product) — The Deployable Decision Unit

What it is

A **Ruleset** is a **decision product**: a cohesive package of rules logic that answers *one business question*.

Example:

- TravelDelayAdjudicationRuleset

What it contains (conceptually)

- many rules
- grouped into phases (rule groups)
- possibly generated from templates or decision tables
- associated parameters (thresholds per region/product)
- metadata: owner, region scope, effective dates, approval state

Versioning

Rulesets MUST be versioned because they represent immutable decision logic used for audit/replay:

- TravelDelayAdjudicationRuleset v2.1.0

Relationship

- **Decision → Ruleset:** 1-to-many (sometimes 1-to-1)
- **Ruleset → Ruleset Version:** 1-to-many
- **Ruleset Version → Executable Artifact (KJAR):** 1-to-1 (recommended)

Key principle: **rules are reusable at design time**, but each *ruleset version* freezes a snapshot for runtime.

3) Ruleset Version — The Frozen Snapshot (Audit Anchor)

What it is

A specific immutable snapshot of a ruleset:

- exact DRL content
- exact templates/decision tables resolved
- exact parameter versions (if externalized)
- exact dependencies

Why it matters

When someone asks “*which logic was used?*” the answer is:

- ruleset name + version (+ artifact hash)

Relationship

At runtime you log:

- rulesetVersion
- which rule IDs fired
- optionally the rule content hash(es)

This is your **audit + replay** foundation.

4) Rule — The Reusable Unit of Logic (But Not a Deployable Unit)

What it is

A **Rule** is an atomic piece of logic:

- IF conditions match facts
- THEN produce decision evidence / actions

Identity vs Version

Rule ID should be **stable** across edits if the business intent is the same:

- POLICY_ACTIVE_ON_LOSS_DATE

Do not treat rule versioning as a deployment mechanism.

Instead:

- rule revision can exist as metadata (commit hash / revision stamp)
- but **ruleset version** is the governance version

Reuse across rulesets

This is the point you raised (correctly):

- A rule like “Policy must be active on date of loss” is used in many decisions.
- Therefore: **Rule has no intrinsic rulesetId**.
- Association is **many-to-many** via a composition record:

RulesetComposition (join)

- rulesetId
- ruleId
- ruleGroup / phase
- enabled flag
- optional overrides

Relationship

- Rule Library → many Rulesets
 - Ruleset Version → contains a compiled snapshot of rules
-

5) Rule Group — The Execution Phase (Order + Structure)

What it is

A **Rule Group** is not a “business asset”—it’s an **execution structuring mechanism**.

Its purpose:

- make rules maintainable
- prevent salience wars
- enforce phased evaluation (validation → eligibility → calc → routing)

In Drools terms

Rule groups typically map to:

- agenda-group (most common)
- ruleflow-group (if orchestrated via BPMN / process)

Relationship

- Ruleset contains multiple Rule Groups
- Rule belongs to exactly one group *within a ruleset* (best practice)
- Same rule reused in another ruleset can be placed into a different group (often still “validation”)

Rule groups define “how to evaluate” *inside* a ruleset.

They do not define “what decision” (that’s ruleset).

6) Rule Template — Logic Pattern with Parameters (Scaling Mechanism)

What it is

A template is a **rule generator pattern**:

- stable DRL structure
- placeholders for variable content
- driven by parameter rows (CSV/DB)

Used when:

- the logic is consistent
- thresholds vary by region, product, partner, coverage, etc.

Relationship

Templates generate **many rules** that end up inside a ruleset version.

So:

- **Template** (pattern) → produces **Rules** (instances)
- **Rules** → organized into **Rule Groups**
- **Ruleset Version** → freezes generated rules

Why templates matter for global STP

They enable:

- EMEA/APAC/LATAM differences as parameters, not copy-pasted rules
- faster rollout
- consistent governance

7) Decision Table — Business-Friendly Bulk Rule Authoring

What it is

A **Decision Table** is a structured tabular way to define many rules.
Drools supports decision tables (often XLS/CSV) that compile to rules.

Use when:

- business wants to manage thresholds/conditions in a tabular form
- rule logic fits a “row = rule” mental model

Avoid when:

- logic requires complex joins, temporal constraints, or deep object graph reasoning
- you need advanced control flow or multi-step reasoning

Relationship

Decision tables are another **rule source**:

- `DecisionTable` → compiled → Rules → grouped → packaged → executed

Think:

- Templates and decision tables are both *rule factories*
 - DRL is direct hand-coded rules
-

8) KIE / Drools Runtime Chain (Execution Hierarchy)

This is the “real hierarchy” in Drools runtime.

8.1 KJAR (Knowledge JAR)

A **versioned deployable artifact** containing:

- DRL
- decision tables
- templates
- rule assets
- `kmodule.xml`
- dependencies

This is what your decision service loads from artifact storage.

In enterprise STP: treat **KJAR as the immutable runtime representation of a ruleset version**.

8.2 KIE Module

The internal representation of a set of knowledge bases and sessions defined in `kmodule.xml`.

8.3 KIE Base (KieBase)

A compiled, in-memory representation of:

- rules
- processes (if any)
- type declarations

A KIE Base is relatively heavy to build; you usually:

- cache it per ruleset version
- refresh on version changes

8.4 KIE Session (KieSession / StatelessKieSession)

The actual execution context:

- Working memory (facts)
- Agenda (what can fire)
- Globals (carefully used)
- Event listeners (for audit)

For STP you typically prefer:

- StatelessKieSession for high volume deterministic decisions

8.5 Facts (Working Memory)

The objects inserted for evaluation:

- Claim facts
- Policy facts
- Derived facts (e.g., “PolicyIsActive”)
- Reference facts (jurisdiction configs, thresholds, etc.)

Facts are the boundary between your platform and Drools:

- keep them small, stable, and versioned as contracts

8.6 Agenda & Activations

Drools determines which rules match and puts them on the agenda.

Order can be affected by:

- agenda groups
- salience (avoid heavy use)
- ruleflow (if used)

8.7 Fired Rules → Decision Output

Rules should not mutate domain objects; they should write to a **Decision** output (your platform model):

- outcomes
 - reason codes
 - evidence
 - routing decisions
-

9) Other Critical Items You're Missing (Highly Important)

9.1 Fact Model / Canonical Decision Contract

This is the most important piece in STP:

- your facts are your “decision API contract”
- decouple facts from upstream raw payloads (FNOL schemas)
- stabilize facts to protect rules from upstream churn

9.2 Effective Dating (Temporal Validity)

Insurance decisions must support:

- effectiveStart / effectiveEnd of ruleset versions
- policy period alignment
- jurisdiction changes over time

Common pattern:

- choose ruleset version based on **loss date**, not “today”

9.3 Rule Units (Drools feature)

Rule Units can help structure rules + data sources cleanly. Not always necessary, but relevant if you want stronger modularization.

9.4 Listeners & Audit Capture

You must implement:

- rule firing listeners
- capture fired rules
- capture ruleset version
- persist decision trace

Drools doesn't persist this automatically.

9.5 Promotion & Governance Workflow

Your UI implies you already have this, but it must be explicit:

- Draft → Review → Approved → Promoted → Deprecated
- environment constraints (SIT/UAT/PROD)
- approval gates (architecture + business + compliance)

9.6 Artifact Registry (Ruleset Registry)

Store:

- ruleset version → artifact coordinates (groupId/artifactId/version)
- artifact hash
- effective date
- status per env

This is how runtime “knows what to load”.

10) Putting It Together — Relationships Summary

Here's the hierarchy and relationships in one view:

Business / Platform

- **Decision**
 - Ruleset
 - Ruleset Version
 - includes **Rule Groups**
 - contains **Rules**
 - rules may be hand-authored DRL **or generated from**
 - Templates
 - Decision Tables
 - packaged as **KJAR**
 - registered in **Ruleset Registry**
 - executed by **Decision Service**
 - audit persisted in **Decision Trace Store**

Drools / Runtime

- **KJAR** → **KieModule** → **KieBase** → **KieSession** → **Facts** → **Agenda** → **Fired Rules** → **Decision Output**

11) The Most Important “Insurance STP” Rule of Thumb

Reuse rules at design time via a Rule Library.
Freeze rules at runtime via Ruleset Versions (KJARs).

That gives you:

- reuse without drift
 - auditability
 - replay
 - safe global rollout
-

12) Practical Guidance for Your UI + DB Model

Given your UI (“products, rules, rulesets”), the clean model is:

Tables / Entities

- Product
- Decision
- Ruleset
- RulesetVersion
- RuleLibrary (Rule)
- RulesetComposition (join)
- TemplateLibrary
- DecisionTableLibrary
- ParameterSet (optional)
- ArtifactRegistry
- ApprovalWorkflow / Status

Runtime resolution

- Product + Decision + Loss Date + Region → choose RulesetVersion → load KJAR → run Drools