ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 13 Issue 05 May 2026

A Comparative Performance Study of Procedural Rule Evaluation vs Rete-Based Drools Engine at Enterprise Level

Sangeeta Manna

Master of Computer Applications, UTech. WB. India
Senior Software Programmer in Banking, Insurance & Service-related IT Industry
Email: sangeeta.gt.yt@gmail.com

Abstract— *This case study presents a controlled experimental comparison between a custom procedural pricing engine and a Rete-based rule engine (Drools) under high-volume policy evaluation workloads at enterprise level. Using JMH-based micro benchmarking on an Intel i5-12700H system (16GB RAM, JDK 17), we evaluated performance across rule sets ranging from 50 to 500 insurance rules and policy volumes of 100,000 and 500,000 records.*

Results demonstrate that procedural insurance rule evaluation exhibits near-linear scaling with respect to rule count and maintains throughput stability across increasing policy volumes. However, Drools incurs significant overhead for flat rule sets due to Rete network construction and fact propagation costs. The findings highlight architectural trade-offs between deterministic procedural engines and declarative rule-based systems in high-throughput environments.

Index Terms— *JMH (Java Microbenchmark Harness), RAM (Random Access Memory), BRMS (Business Rule Management System), JDK (Java Development Kit), DRL (Drools Rule language)*

I. INTRODUCTION

Rule-based decision engines play a critical role in modern enterprise systems, enabling automated processing for applications such as insurance pricing, fraud detection, credit scoring, regulatory validation, and policy underwriting. As data volumes increase and real-time decision requirements become more stringent, the performance and scalability of rule evaluation mechanisms become central architectural concerns.

Two primary paradigms dominate the implementation of rule engine. The first is procedural rule evaluation, where rules are encoded directly in deterministic program logic, offering predictable performance and low runtime overhead. The second is declarative rule processing using Rete-based [1] engines such as Drools, which construct optimized discrimination networks to efficiently match patterns across large rule sets, particularly in scenarios involving shared conditions and complex fact relationships. Drools is a Business Rule Engine (BRE) that let us write business logic as rules instead of hard-coded logic.

This study empirically evaluates the performance characteristics of both approaches under large-scale workloads, analysing their scalability, throughput behaviour, and architectural trade-offs to inform system design decisions in high-volume enterprise environments.

II. PROBLEM STATEMENT

Enterprise systems increasingly depend on rule-based engines to automate high-volume decision-making processes such as insurance pricing, underwriting, fraud detection, and compliance validation. While declarative rule engines like Drools (based on the Rete algorithm) provide flexibility and maintainability, procedural rule implementations often claim superior runtime performance. However, there is limited empirical evidence quantifying the performance trade-offs between these two paradigms under large-scale workloads.

The core problem addressed in this case study is:

How do procedural rule engines and Rete-based rule engines compare in terms of scalability, throughput, memory behaviour, and computational efficiency when evaluating large rule sets across high policy volumes?

III. AIMS AND OBJECTIVES

The primary aim of this research is to empirically analyse architectural trade-offs between procedural and Rete-based drools engines.

Objectives:

- Measure execution time growth with increasing rule counts.
- Evaluate scalability across large policy volumes in industry level.
- Compare throughput efficiency between architectures.
- Identify performance stability patterns.
- Provide architectural decision guidance for enterprise systems.

IV. METHODOLOGY & DESIGN

This research follows an experimental benchmarking methodology using Java Microbenchmark Harness (JMH) [4] to ensure controlled and statistically reliable measurements.

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 13 Issue 05 May 2026

1. Environment and Experimental Setup

1.1. Software Design

- JDK 17, Spring Boot
- JMH [4]
- Build Tool – Maven

1.2. Hardware Design

- Intel i5-12700H, 16GB RAM
- Single-threaded execution (baseline)

1.3. Workload Design

- Policy volumes: 100,000 and 500,000
- Rule sets: 50, 150, 300, 500

2. Implementation Design

Two systems were implemented:

A. Procedural Engine

- Deterministic rule evaluation using code-based conditions.
- Time complexity approximates O(N × R).

B. Drools Engine

- Dynamic DRL generation.
- Rete-based pattern matching.
- Batch insertion of facts followed by rule firing.

3. Drools and the Rete Algorithm Design

Now we move to the architectural core of the rule engine: the internal working model that enables high-performance rule evaluation at scale. Drools is a declarative Business Rule Management System [6] that implements an optimized variant of the Rete algorithm known as PHREAK, designed to improve lazy evaluation and reduce unnecessary fact propagation. Instead of evaluating rules sequentially through traditional conditional logic, Drools builds an optimized network structure that minimizes redundant computations.

3.1 Introduction to Rete Algorithm

Rete is a pattern-matching algorithm designed to efficiently evaluate large numbers of rules against dynamic fact sets. Instead of evaluating every rule against every fact—an approach that typically results in procedural time complexity of O(N × R), where N represents the number of facts and R the number of rules—the Rete algorithm adopts a fundamentally different strategy. It constructs a discrimination network that organizes rule conditions into a structured graph, allowing common condition nodes to be shared across multiple rules. This shared architecture eliminates duplicate evaluations of identical patterns. Furthermore, rather than reprocessing the entire rule set whenever data changes, Rete propagates only the modified or newly inserted facts through the network. By maintaining intermediate results and avoiding redundant evaluations, the algorithm significantly improves efficiency, especially in large-scale, dynamic environments where both the number of rules and facts can grow substantially.

3.2 Internal work pattern of Rete

Internally, Drools constructs a structured Rete network composed of different types of nodes, each responsible for a specific stage of rule evaluation. The first layer consists of **alpha nodes**, which perform single-fact condition checks. These nodes filter incoming facts based on simple constraints, such as field comparisons or attribute validations.

Beyond the alpha layer are **beta nodes**, which handle joins between multiple facts. Beta nodes combine partial matches received from alpha nodes (or other beta nodes) and evaluate relational conditions between them. This enables the engine to process complex rule patterns that depend on multiple facts simultaneously. At the final stage are **terminal nodes**, which represent completed rule conditions. When all required conditions for a rule are satisfied, the flow reaches a terminal node, indicating that the rule is eligible for execution.

The operational flow begins when facts are inserted into the working memory. These facts propagate through the alpha network, where initial filtering occurs. Matching partial conditions are stored and reused, allowing the engine to avoid re-computation. The filtered results then move into beta nodes, where joins between related facts are evaluated. Once all rule conditions are satisfied, a rule activation is created and placed on the agenda. Finally, the agenda manages and fires the matched rules according to the configured conflict resolution strategy.

3.3 Reasons for the slower performance of Drools compared to flat rule scenarios

In this experiment:

- Rules are independent
- No fact joins
- No condition sharing
- No incremental updates

Therefore:

- Drools still build Rete network
- Drools still propagate 500,000 facts
- Drools maintains agenda

But it gains no structural benefit. Thus, overhead dominates. This is expected and correct behaviour.

This does not imply that Drools is inefficient, that the Rete algorithm is flawed, or that procedural rule evaluation is universally superior. Rather, it highlights an important contextual distinction. The optimizations offered by the Rete algorithm become truly advantageous in scenarios where rules share common conditions, multiple fact types interact in complex ways, facts are updated incrementally, and large rule interaction graphs exist. In such environments, Rete's ability to reuse partial matches and avoid redundant evaluations provides measurable performance benefits. However, if an experiment is conducted using relatively flat rule structures without shared conditions or complex interdependencies, those architectural advantages may not surface.

4. Performance metrics captured:

- Average execution time (ms)
- Throughput (policies/sec)
- Stability across scaling

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 13 Issue 05 May 2026

V. V. RELATED WORKS & IMPLEMENTATION

This experiment has focused on improving rule engine scalability, memory efficiency, and incremental update handling. Modern enterprise rule engines such as **Drools** implement enhanced variants of the Rete algorithm (e.g., ReteOO, PHREAK) to support object-oriented fact models and optimized agenda management. PHREAK [3] improves classical Rete by introducing segmented evaluation and lazy propagation, reducing redundant computation in complex rule interaction scenarios. These systems are widely adopted in financial services, insurance underwriting, fraud detection, and compliance automation due to their declarative structure and dynamic rule management capabilities.

Implementation

To enable controlled comparison, two independent rule evaluation systems were implemented:

1. Procedural Rule Engine

A custom procedural engine was developed using deterministic conditional logic. Rules were programmatically generated to simulate underwriting and pricing logic across multiple policy attributes. Each rule directly evaluated policy fields and adjusted premium values accordingly.

Key characteristics:

- Direct in-memory evaluation
- No intermediate network construction
- Time complexity approximates O(N × R)
- Minimal runtime allocation overhead

This engine serves as the baseline deterministic model for performance comparison.

Code Snippet for Procedural Engine:

```
private void generateRules(int ruleCount) {  1 usage   sangeeta *
    for (int i = 0; i < ruleCount; i++) {
        int type = i % 8;
        switch (type) {
            case 0 -> rules.add(p -> {
                if (p.getAge() > 60)
                    p.setFinalPremium(p.getFinalPremium() + 200);
            });
            case 1 -> rules.add(p -> {
                if (p.isSmoker())
                    p.setFinalPremium(p.getFinalPremium() + 300);
            });
            case 2 -> rules.add(p -> {
                if (p.isDiabetic())
                    p.setFinalPremium(p.getFinalPremium() + 250);
            });
            case 3 -> rules.add(p -> {
                if (p.getCreditScore() < 600)
                    p.setFinalPremium(p.getFinalPremium() + 400);
            });
```

2. Drools-Based Rule Engine

A Drools engine implementation was constructed using dynamically generated DRL (Drools Rule Language) files. Rule sets of varying sizes (50–500 rules) were programmatically generated and compiled at runtime using KieHelper. Policies were inserted into a KieSession, and rules were fired in batch mode.

Key characteristics:

- Rete-based [1] pattern matching
- Working memory and agenda management
- Fact insertion and propagation through alpha-beta network
- Increased structural and allocation overhead

The Drools implementation allows evaluation of how Rete-based optimization behaves under flat, independent rule scenarios.

Code Snippet for Drools Engine:

```
14      private final KieBase kieBase;  3 usages
15
        Edit | Explain | Test | Document | Fix
16      public DroolsPricingEngine(int ruleCount) {  1 usage   sangeeta
17          String drl = DrlGenerator.generate(ruleCount);
18          KieHelper helper = new KieHelper();
19          helper.addContent(drl, ResourceType.DRL);
20          Results results = helper.verify();
21          if (results.hasMessages(Message.Level.ERROR)) {
22              throw new RuntimeException(results.getMessages().toString());
23          }
24          this.kieBase = helper.build();
25      }
```

Code Snippet fir Dynamic Drools Generator

```
    Edit | Explain | Test | Document | Fix
    public static String generate(int ruleCount) {  1 usage   sangeeta
        StringBuilder drl = new StringBuilder();
        drl.append("package rules;\n");
        drl.append("import com.research.model.Policy;\n\n");
        for (int i = 1; i <= ruleCount; i++) {
            drl.append("rule \"Rule_").append(i).append("\"\n");
            drl.append("when\n");
            drl.append("    $p : Policy(age > ").append(20 + (i % 50)).append(")\n");
            drl.append("then\n");
```

Experimental Control

To ensure consistency:

- Java Microbenchmark Harness was used.
- Warm-up and measurement iterations were applied.
- Single-threaded execution ensured deterministic baseline comparison.
- Identical policy datasets were used for both engines.

This controlled implementation enables direct architectural comparison under identical workload conditions.

VI. RESULT ANALYSIS

The experimental results demonstrate the following:

1. Linear Rule Scaling in Procedural Engine

Execution time increases proportionally with rule count, confirming O(N × R) behaviour.

2. Throughput Stability Across Policy Volumes

For a fixed rule count, throughput remains nearly constant when scaling from 100,000 to 500,000 policies, indicating CPU-bound deterministic performance.

3. Drools Overhead in Flat Rule Scenarios

In independent rule configurations, Drools exhibits higher execution time due to:

- Rete network construction
- Fact propagation overhead
- Agenda management costs

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 13 Issue 05 May 2026

4. Architectural Implication

Procedural engines outperform Rete-based systems in flat, independent rule scenarios. However, Rete engines are expected to provide benefits in:

- Shared condition reuse
- Multi-fact joins
- Incremental updates

These findings highlight that architecture selection must align with rule complexity rather than assumed performance advantages.

5. Scaling Analysis

5.1 Rule Scaling Behaviour

The procedural engine exhibits near-linear time growth with rule count.

Time complexity approximates: O(N × R)

Where: N = number of policies, R = number of rules

Throughput decreases proportionally as rule count increases.

5.2 Policy Volume Scaling

Throughput remains nearly constant across policy volumes for the same rule count:

Example:

At 300 rules:

- 100k -> 904k/sec
- 500k -> 902k/sec

This indicates CPU-bound deterministic behaviour with minimal memory degradation

6. JMH Benchmark Configuration

To ensure statistically reliable and reproducible performance measurements, the Java Microbenchmark Harness (JMH) was used. JMH [4] is the official benchmarking framework developed by OpenJDK and is specifically designed to avoid common micro benchmarking pitfalls such as dead-code elimination, JVM warm-up bias, and compiler optimizations.

6.1 Benchmark Mode

The benchmarks were executed using:

- Mode: Average Time (Mode.AverageTime)
- Output Time Unit: Milliseconds (ms)
- Threads: 1 (Single-threaded baseline)
- Forks: 3
- Warm-up Iterations: 5
- Measurement Iterations: 5
- Warm-up Time per Iteration: 10 seconds
- Measurement Time per Iteration: 10 seconds

Command used:

java -jar jmh-benchmark.jar -f 3 -wi 5 -i 5

6.2 State Scope

The benchmark class was annotated with: @State(Scope.Thread)

This ensures that each benchmark thread maintains its own isolated state, preventing unintended shared memory effects.

```
Edit | Explain | Test | Document | Fix
@BenchmarkMode(Mode.AverageTime)  no usages  sangeeta
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@State(Scope.Thread)
public class PricingBenchmark {

    @Param({"50", "150", "300", "500"})  2 usages
    private int ruleCount;
```

6.3 Dead Code Elimination Prevention

To avoid compiler optimizations removing unused computations, JMH's Blackhole mechanism was used:

```
// Drools Batch Benchmark
@Benchmark  no usages  sangeeta *
public void benchmarkDroolsEngine(Blackhole bh) {
    droolsEngine.evaluateBatch(policies);
    bh.consume(policies.get(0).getFinalPremium());
}
```

6.4 Controlled Setup Phase

All policy generation and engine initialization were performed in: @Setup(Level.Trial)

This ensures:

- Data generation is excluded from measurement
- Only rule evaluation logic is benchmarked
- Compilation and DRL building occur once per trial

6.5 Benchmark Integrity Measures

To increase reliability:

- Identical policy datasets were used across engines.
- GC behaviour was stabilized via warm-up.
- Single-thread execution eliminated synchronization bias.
- No I/O or logging was included inside benchmark methods.
- Results were averaged across forks to reduce JVM randomness.

Data Analysis for Procedural Engine

Manual Baseline Table

Experiment	Policies	Rules	Manual Baseline -Avg Time (ms)	JMH environment - Avg Time(Ms)
1	1,00,000	50	25.753300000000003 ms	aprox-100 ms
2		150	62.83368 ms	~300 ms
3		300	121.56624000000002 ms	~565 ms
4		500	199.78492 ms	~950 ms
1	5,00,000	50	121.73142 ms	~103 ms
2		150	296.28159999999997 ms	~900 ms
3		300	576.3886400000001 ms	581.72 ms
4		500	986.7476800000001 ms	924.32 ms

Experiments conducted on Intel i7-12700H, 16GB RAM, JDK 17.

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 13 Issue 05 May 2026

Data Analysis for Drools Engine

JMH environment-Approx Throughput (policies/sec)	PricingBenchmark- Avg Time(Ms)	PricingBenchmark -Throughput (policies/sec)
~5,000,000	20.38	49,07,000
~1,666,000	57.41	17,41,000
~885,000	110.57	9,04,300
~526,000	182.33	5,48,400
~4,854,000	156.8	31,88,000
~555,000	296.3	16,89,000
~859,490	554.5	9,02,000
~540,930	915.2	5,46,000

VII. FUTURE WORK

Future research will extend this study in the following directions:

- Multi-threaded performance analysis to evaluate concurrency scaling.
- Memory and garbage collection profiling to analyse allocation behaviour.
- Complex rule interaction experiments including multi-fact joins and shared conditions.
- Incremental update benchmarking to assess Rete optimization benefits.
- Hybrid architecture exploration combining procedural cores with declarative rule overlays.

Such extensions will provide a more comprehensive performance model for enterprise decision engine design.

VIII. CONCLUSION

This research demonstrates that rule engine performance is not universally determined by technology choice, but by architectural alignment with workload complexity. Through large-scale empirical benchmarking, the study shows that procedural rule evaluation delivers superior throughput in flat, high-volume scenarios, while Rete-based engines like Drools introduce structural overhead that is justified only in complex, dynamic rule ecosystems.

The key insight is that architecture must be workload-aware. Selecting the wrong rule paradigm can result in significant performance inefficiencies at scale. This work provides data-driven guidance for designing high-throughput decision systems in insurance, finance, and regulatory domains, reinforcing the importance of performance engineering in enterprise architecture.

REFERENCES

[1] C. L. Forgy. “Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.” Artificial Intelligence, vol. 19, no. 1, 1982, pp. 17–37.

[2] Red Hat. “Drools Documentation – Business Rules Management System.” Red Hat Documentation. Available: https://docs.drools.org/

[3] Red Hat. “Drools PHREAK Algorithm Description.” Drools Official Documentation. Available: https://docs.jboss.org/drools/

[4] A. Shipilev. “Java Microbenchmark Harness (JMH) Project.” OpenJDK Documentation. Available: https://openjdk.org/projects/code-tools/jmh/

[5] M. Bali. Drools JBoss Rules 5 Developer’s Guide. Packt Publishing, 2009.

[6] P. Harmon. Business Rule Management and Service-Oriented Architecture. Morgan Kaufmann, 2007.

[7] T. Doorenbos. “Production Matching for Large Learning Systems.” Carnegie Mellon University, 1995.