J-Sentinel

Overview

J-Sentinel is a production-grade, extensible static code analysis platform tailored for Java applications. It leverages graph-based modeling, taint tracking, and flow-sensitive analysis to detect OWASP Top 10 vulnerabilities and complex logic flaws during development. With its developer-first CLI, robust REST backend, scalable analysis pipeline, and intuitive dashboards, J-Sentinel seamlessly integrates into CI/CD workflows and developer tooling.

1. Architectural Breakdown

J-Sentinel adopts a modular, microservices-based architecture designed for scalability, security, and deep code analysis. The system is divided into five core layers, each with distinct responsibilities:

1.1 Developer Interaction Layer

- **Components**:
  - **IDE/CLI Integration**: Developers interact via the J-Sentinel CLI (Java-based) or IDE plugins.
  - **CI/CD Pipelines**: Integrates with GitHub/GitLab webhooks to trigger scans during pull requests or commits.
- **Key Workflow**:
  - Code is scanned locally or via CI tools.
  - The CLI extracts code structure (AST, control/data flow) and uploads a graph model to the backend.

1.2 API Gateway Layer

- **Component**:
  - **REST API Gateway (Spring Boot)**: Manages authentication (JWT, OAuth2), rate limiting, and audit logging. Routes scan data to downstream services.
- **Security**: Role-Based Access Control (RBAC) ensures only authorized users access scan results or rule configurations.

1.3 Analysis & Processing Layer

- **Core Modules**:
  - **Static Analyzer Engine**:
    - **Technologies**: JavaParser (AST generation), JGraphT (in-memory CFG/DFG).
    - **Function**: Performs taint propagation to track unsafe data flows (e.g., user input → sensitive sink).
  - **Graph Processor**:
    - **Technologies**: Neo4j (graph database), Cypher queries.
    - **Function**: Models code logic (e.g., input → function → sink) and executes vulnerability detection via rule-based Cypher queries.
  - **Rule Engine**:
    - **Technologies**: Custom DSL (YAML) + Drools (optional).
    - **Function**: Encapsulates OWASP Top 10 rules and business-specific logic. Translates DSL to Cypher for Neo4j.

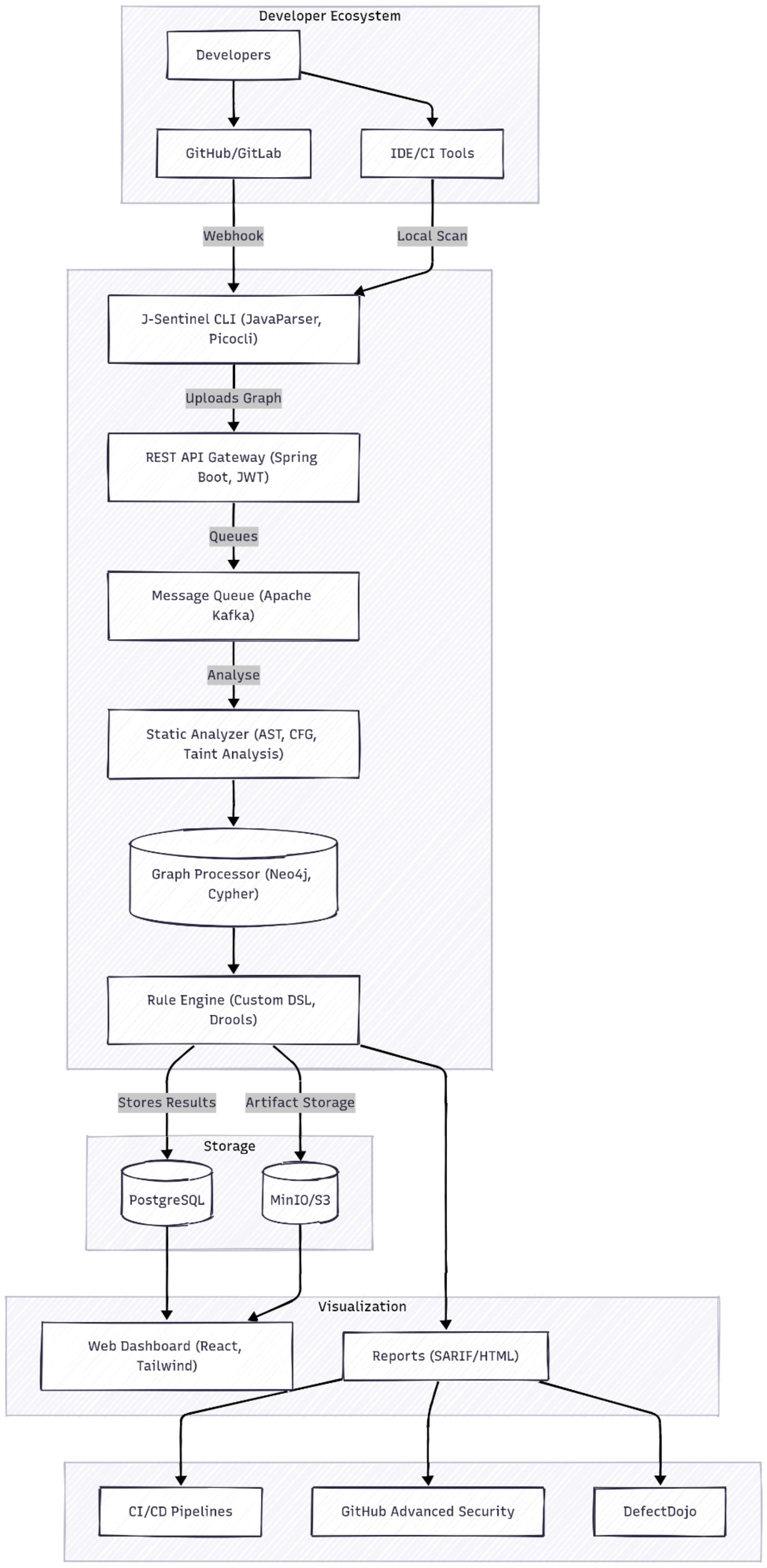
1.4 Storage Layer

- **Components**:
  - **PostgreSQL**: Stores metadata (users, projects, scan history) and rule configurations.
  - **MinIO/S3**: Manages raw scan artifacts (e.g., code graphs, SARIF reports).
- **Scalability**: MinIO/S3 supports distributed storage for large enterprises.

1.5 Visualization Layer

- **Component**:
  - **Web Dashboard (React + TailwindCSS)**:
    - **Features**: Interactive visualization of taint paths (Cytoscape.js/D3.js), scan history, and rule management.
    - **Integration**: Renders SARIF/HTML reports and exports findings to GitHub Advanced Security or DefectDojo.

The diagram below illustrates the high-level architecture of the J-Sentinel Static Analysis Platform. It outlines the complete flow from developer interactions—through CLI scanning and RESTful API communication—to advanced static analysis, graph processing, and result visualization. Each core component, including the analyzer engine, graph processor, rule engine, and UI dashboard, is modularized to ensure scalability, security, and extensibility. The architecture supports modern DevSecOps pipelines and integrates seamlessly with CI/CD systems, enabling continuous vulnerability detection across Java applications.


Developer Ecosystem
Developers
GitHub/GitLab
IDE/CI Tools
Webhook
Local Scan
J-Sentinel CLI (JavaParser, Picocli)
Uploads Graph
REST API Gateway (Spring Boot, JWT)
Queues
Message Queue (Apache Kafka)
Analyse
Static Analyzer (AST, CFG, Taint Analysis)
Graph Processor (Neo4j, Cypher)
Rule Engine (Custom DSL, Drools)
Stores Results
Artifact Storage
Storage
PostgreSQL
MinIO/S3
Visualization
Web Dashboard (React, Tailwind)
Reports (SARIF/HTML)
CI/CD Pipelines
GitHub Advanced Security
DefectDojo

2. Tech Stack Analysis

The stack is chosen for performance, extensibility, and alignment with Java ecosystems.

Layer	Technologies	Rationale
Language & Parsing	Java, JavaParser	Native support for Java AST/CFG extraction; JavaParser is industry-standard.
Backend	Spring Boot, JWT	Spring Boot simplifies REST API development; JWT ensures stateless auth.
Graph Processing	Neo4j, Cypher	Optimized for graph traversals (e.g., detecting multi-hop vulnerabilities).
Storage	PostgreSQL, MinIO	PostgreSQL for ACID compliance; MinIO for S3-compatible object storage.
Frontend	React, TailwindCSS, Cytoscape.js	React enables dynamic UIs; Cytoscape.js visualizes complex code graphs.
Scalability	Kafka, Kubernetes	Kafka decouples scan processing; Kubernetes enables horizontal scaling.

Layer	Technologies	Rationale
Security	OAuth2, RBAC, Input Sanitization	Protects against injection attacks and unauthorized access.

3. End-to-End Workflow

3.1 Code Submission & Scan Trigger

1. **Trigger**: Code is pushed to GitHub (webhook) or scanned locally via CLI.
2. **CLI Execution**:
    - Extracts AST, control flow, and taint flow using JavaParser.
    - Generates a code graph and uploads it to the REST API Gateway.

3.2 Static Analysis & Graph Modeling

1. **Static Analyzer Engine**:
    - Builds AST and Control Flow Graph (CFG).
    - Identifies taint sources (e.g., user inputs) and propagates them to sinks (e.g., SQL queries).
2. **Graph Processor**:
    - Persists the code graph in Neo4j.
    - Executes Cypher queries to detect rule violations (e.g., SQLi, XSS).

3.3 Rule Evaluation & Reporting

1. **Rule Engine**: Applies custom DSL or Drools rules to flagged paths.
2. **Result Formatting**:
    - Findings are formatted into SARIF (for GitHub integration), JSON, or HTML.
    - Reports are stored in PostgreSQL (metadata) and MinIO (raw files).

3.4 Visualization & Action

1. **Dashboard**:
    - Developers view taint paths and rule violations in an interactive graph.
    - Adjust rules or rescan via the UI.
2. **CI/CD Integration**: Findings are pushed to GitHub Advanced Security or DefectDojo.

4. Scalability & Reliability

- **Concurrent Scans**: Kafka queues manage high throughput; Kubernetes scales API/graph pods.
- **Large Projects**: Code is parsed in chunks; parallel AST generation reduces latency.
- **Multi-Tenancy**: PostgreSQL schema isolation secures tenant data.
- **Backup**: MinIO versioning and PostgreSQL snapshots ensure data integrity.

5. Strengths & Considerations

- **Strengths**:
  - **Graph-Based Analysis**: Neo4j enables deep path analysis for complex vulnerabilities.
  - **Extensible Rules**: Custom DSL allows teams to add context-specific security checks.
  - **DevSecOps Integration**: SARIF support and CI hooks embed security early in SDLC.
- **Considerations**:
  - **Java-Centric**: Limited to JVM languages (roadmap includes Kotlin/Scala).
  - **Neo4j Licensing**: Community edition may lack enterprise-grade features.

6. Conclusion

J-Sentinel's architecture combines modern technologies (Spring Boot, Neo4j, React) with graph-based analysis to address critical gaps in Java static analysis. Its modular design ensures scalability, while SARIF and CI/CD integrations align with DevSecOps practices. Future enhancements (ML, IDE plugins) position it as a forward-looking platform for enterprise code security.