

# **Technical Architecture Document:**

## **Intelligent Service Mesh**

## 1. Executive Summary

The goal is to build a high-concurrency **API Gateway** using **Java 21 Virtual Threads** that manages traffic between upstream consumers and downstream mock services. The system uses **Resilience4j** for fault tolerance, **Kafka** for real-time telemetry, and **Redis** for dynamic routing configuration. A local **Ollama LLM** acts as an advisory engine, suggesting routing optimizations based on historical health data.

## 2. System Architecture

The system follows a decoupled, event-driven design to ensure the gateway remains functional even if secondary components (AI, Kafka) are offline.

## 3. Core Tech Stack

Component	Technology	Purpose
Language	Java 21	Leverage <b>Virtual Threads</b> for massive I/O scale.
Framework	Spring Boot 3.4	Core container and auto-configuration.
API Layer	Netflix DGS	GraphQL-based federation and typed schema.
Gateway	Spring Cloud Gateway	Path-based routing and filter chains.
Reliability	Resilience4j	Circuit breakers, Rate limiters, and Retries.
Data Store	Redis	Dynamic routing tables and Rate-limit quotas.
Messaging	Apache Kafka	Async metrics streaming for AI analysis.
AI Engine	Ollama (Llama 3.2)	Advisory engine via <b>Spring AI</b> .

## 4. Implementation Workflow & Approach

## Phase 1: The Resilient Foundation

- **Virtual Thread Configuration:** Enable `spring.threads.virtual.enabled=true` to ensure the gateway is non-blocking.
- **Resilience4j Integration:** Wrap every route in a **Circuit Breaker**.
  - *Approach:* Define a `fallbackMethod` for each route that serves cached data from Redis if the backend service is down.
- **Dynamic Routing:** Instead of hard-coding routes in `application.yml`, we will use a `RouteDefinitionRepository` that reads from **Redis**. This allows "Hot-Reloading" of routes.

## Phase 2: The Telemetry Pipeline

- **Gateway Filters:** Create a custom global filter that captures `startTime`, `endTime`, `HttpStatus`, and `ServiceID`.
- **Kafka Producer:** Push these metrics as JSON events to a `gateway-metrics` topic.
- **Mock Services:** Build 3 lightweight Spring Boot apps that simulate "Success," "Latency," and "Random Failure" to test the circuit breakers.

## Phase 3: The Intelligence Layer

- **Spring AI + Ollama:** Create a service that "consumes" Kafka metrics, summarizes the last 5 minutes of data, and sends a prompt to Ollama.
- **Advisory Mode:** Ollama returns a recommendation (e.g., "*Service B is 40% slower than usual; recommend reducing rate limit to 50 req/sec*").
- **Dashboard (The "Hook"):** A simple React/Next.js UI using **DGS GraphQL Subscriptions** to show live traffic lights (Green/Yellow/Red) for each service.

## Phase 4: Automated CI/CD (Final Polish)

- **GitHub Actions CI:** Automate a pipeline that triggers on every push to compile the code (Java 21), run unit tests, and build a **Docker** image.
- **Docker Registry:** Automatically push versioned images to **Docker Hub** or **GitHub Container Registry (GHCR)**.
- **Automated Deployment:** Configure the pipeline to SSH into your instance, pull the latest image, and restart the container to ensure the "instance" is always up-to-date.

---

## 5. "Anti-Fragile" Design Decisions

- **Retry Storm Protection:** Retries will use **Exponential Backoff with Jitter** (built into Resilience4j) to avoid DDOSing our own services during recovery.

- **Fail-Safe Defaults:** If Redis is unavailable, the system will fall back to an in-memory Map of "Critical Routes."
- **Human-in-the-Loop:** The AI will update a suggested\_config key in Redis, but it will not update the active\_config unless a manual toggle is flipped in the UI.