

Adaptive Load Balancing in Java-Based Microservices Using Real-Time Metrics

Vinayaka Hegde

Department of Computer Science and Engineering

Texas A&M University

`vinayaka.hegde@tamu.edu`

Advisor: Dr. Duncan M. (Hank) Walker

Associate Dean for Graduate Programs

`d-walker@tamu.edu`

December 12, 2025

Abstract

Modern cloud-native systems are increasingly adopting microservices architectures for their scalability, modularity, and resilience. However, conventional load balancing strategies—such as round-robin or least-connections—are unable to adapt to runtime variations in service performance, leading to uneven resource utilization and degraded user experience. Prior studies highlight the importance of incorporating real-time observability into load balancing mechanisms to dynamically respond to traffic conditions and avoid routing decisions that overload unhealthy instances [1, 2].

This work presents an adaptive load balancing system implemented within a Java-based microservices framework using Spring Boot and Spring Cloud Gateway. The approach leverages Micrometer to collect real-time latency, error rate, and queue length metrics from each service instance. These metrics are exposed to Prometheus and stored in Redis for high-speed access during routing decisions. A custom gateway filter computes performance scores using a weighted algorithm and routes requests to the most optimal instance in real time. In scenarios where metrics are unavailable, a round-robin fallback ensures continuity. The full system is containerized using Docker and orchestrated with Docker Compose, offering reproducibility, extensibility, and seamless deployment for evaluation and further development.

Keywords: Microservices, Adaptive Load Balancing, Spring Cloud Gateway, Redis, Prometheus, Real-Time Metrics, Docker

1 Introduction

Microservices architectures have transformed how large-scale applications are designed, offering modularity, independent deployment, and horizontal scalability. Despite these benefits, microservices introduce new operational challenges, particularly in request routing and instance load management. Load balancing is critical in this context, as it ensures even distribution of traffic across available service instances. Traditional strategies like round-robin or least-connections are widely used due to their simplicity, but they lack awareness of the dynamic runtime state of service instances. These static methods can inadvertently route traffic to overloaded or unhealthy instances, resulting in increased latency, poor fault tolerance, and inefficient resource utilization [3, 4].

To overcome these limitations, this project proposes an adaptive load balancing mechanism that routes requests based on real-time performance metrics such as latency, error rate, and queue saturation. The system is implemented within a Java-based microservices framework using Spring Boot and Spring Cloud Gateway. Real-time metrics are collected via Micrometer and Prometheus, and stored centrally in Redis for fast access. A custom gateway filter computes a score for each instance and routes traffic to the most responsive service in real time, falling back to round-robin in the absence of valid metrics. The platform is containerized using Docker and Docker Compose for reproducibility. This work forms the foundation for evaluating adaptive load balancing against traditional strategies in terms of responsiveness, scalability, and resilience.

2 Literature Review

Traditional load balancing strategies such as round-robin, least-connections, and random selection are widely used due to their simplicity and low overhead. However, these techniques are static in nature and do not consider the real-time performance of service instances. As a result, they often route traffic to overloaded or degraded instances, leading to performance bottlenecks and reduced fault tolerance in microservices architectures [3].

Recent research has proposed adaptive load balancing mechanisms that dynamically route traffic based on runtime metrics such as latency, queue length, and error rate. For example, Zhang et al. [4] proposed a fault-tolerant microservices architecture that uses real-time metrics and feedback loops to improve routing decisions. Selvakumar et al. [5] implemented a latency-aware load balancer that prioritizes faster instances, significantly reducing average response times under varying loads. These works demonstrate the value of metric-aware traffic distribution in dynamic cloud environments.

From an industry perspective, Linkerd — a popular service mesh — supports adaptive routing using real-time success rates and latency windows to guide decisions [6]. Similarly, Envoy Proxy offers built-in support for dynamic load balancing policies based on upstream host health, request load, and runtime statistics [7]. These systems highlight the growing shift toward intelligent,

metrics-driven routing in production-grade microservices deployments.

This project builds upon these ideas by proposing a lightweight, pluggable adaptive load balancer that integrates seamlessly into a Spring Cloud Gateway setup. Unlike heavyweight service meshes or complex ML-based systems, this approach leverages simple metrics and scoring logic to improve routing decisions without introducing significant system overhead.

3 System Architecture

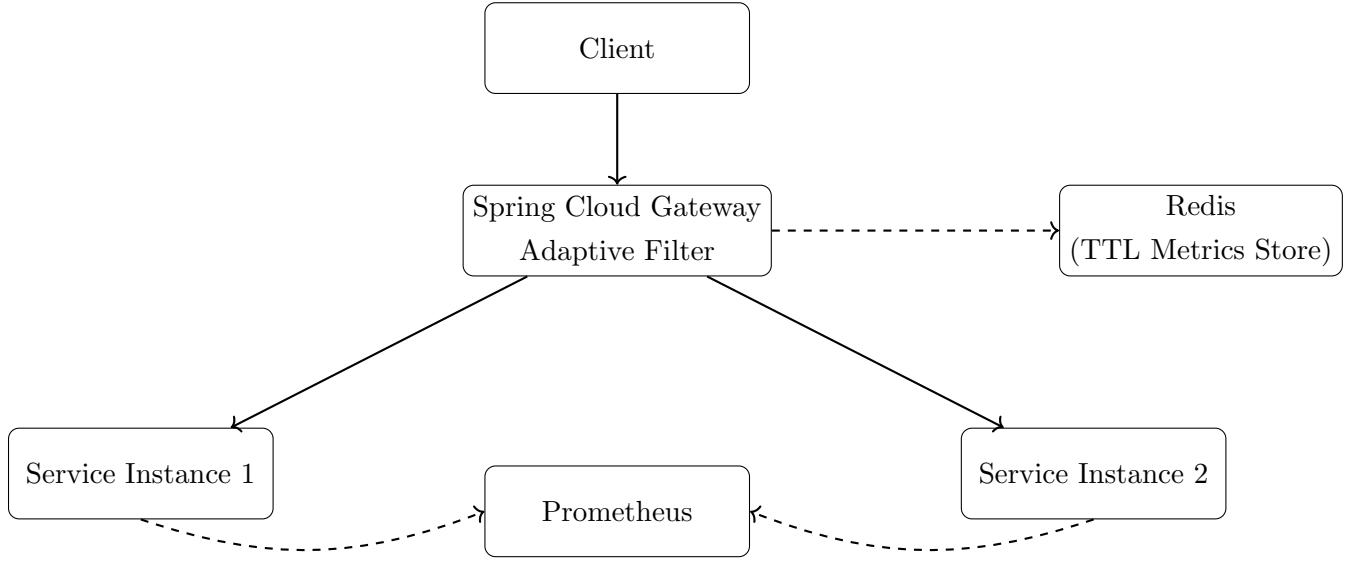


Figure 1: System Architecture of the Adaptive Load Balancer

The proposed system architecture is built around a containerized, metric-driven microservices environment that enables adaptive request routing based on real-time performance data. It is composed of several loosely coupled components that interact through standard HTTP-based protocols and centralized metric sharing. Figure 1 illustrates the high-level structure of the system.

- **Spring Cloud Gateway (SCG)** acts as the reverse proxy and API gateway, sitting at the entry point of the system. A custom filter (`AdaptiveLoadBalancingFilter`) is plugged into SCG to perform dynamic instance selection for every incoming request. This filter replaces traditional routing logic by incorporating real-time performance data into each routing decision.
- **Service Instances** are replicated microservices built using Spring Boot. Each service exposes a REST API and runs independently. Multiple instances of the same service type can be

launched and scaled horizontally. These instances report their health and performance metrics periodically.

- **Micrometer & Prometheus** form the metrics collection subsystem. Each microservice includes the Micrometer library, which exposes runtime statistics via Spring Actuator at the `/actuator/prometheus` endpoint. Prometheus scrapes this endpoint at configurable intervals and stores time-series data on latency, error rates, and request volume.
- **Redis** serves as a high-speed in-memory datastore that holds the most recent snapshot of each instance's performance metrics. This decouples the routing logic from Prometheus, ensuring fast and low-latency access for each request. Redis entries are keyed by service and instance ID, and are automatically expired using a time-to-live (TTL) mechanism to prevent stale routing decisions.
- **Docker and Docker Compose** are used for consistent deployment and orchestration of all components. Docker containers ensure that every microservice, including the gateway, Redis, and Prometheus, runs in an isolated, reproducible environment. Docker Compose manages the container network and startup order, enabling seamless local testing and scaling.

The control flow begins when a client sends a request to the Spring Cloud Gateway. The gateway identifies the target service (based on the route path), retrieves a list of available instances, and queries Redis for the latest performance metrics for each instance. A scoring algorithm then computes a performance score for each instance based on latency, error rate, and queue length. The instance with the highest score is selected, and the request is forwarded to it. If no valid metrics are found for any instance (e.g., during startup or Redis failure), the gateway falls back to a stateless round-robin load balancer.

This modular architecture promotes separation of concerns and can easily be extended. For example, Resilience4j can be integrated to automatically exclude unhealthy instances, or Grafana can be added to visualize performance metrics and routing decisions in real time. The current setup also supports dynamic scaling; new instances can register automatically and begin emitting metrics without requiring gateway reconfiguration.

4 Implementation Details

The system is implemented using Spring Boot and Spring Cloud Gateway, organized into modular Java packages to support maintainability, testability, and separation of concerns. The core logic of adaptive load balancing is encapsulated in a custom gateway filter and is supported by auxiliary components that handle metric retrieval, scoring, fallback routing, and background cleanup.

Project Structure

The codebase is organized into several key packages:

- **filter**: Contains the `AdaptiveLoadBalancingFilter`, a custom Spring Cloud Gateway filter that intercepts each request and routes it based on real-time instance performance.
- **metrics**: Provides classes for managing performance metrics, including the `MetricScoreCalculator` (which calculates scores) and the `RedisMetricRepository` (which interacts with Redis).
- **loadbalancer**: Includes the `RoundRobinLoadBalancer` used as a fallback when metric data is missing.
- **scheduler**: Implements background tasks such as metric cleanup via the `MetricAggregatorScheduler`, which ensures outdated entries in Redis are removed.
- **controller**: Exposes internal routing decisions and metrics via the `DebugController` REST endpoint, useful for debugging and validation.
- **model**: Defines the `InstanceMetrics` data class, which represents metrics such as latency, error rate, and queue length.

Adaptive Load Balancing Filter

The `AdaptiveLoadBalancingFilter` implements the `GatewayFilterFactory` interface and is registered as a Spring bean. When a request is received, it determines the target service based on the route and queries the `MetricRepository` to retrieve the latest `InstanceMetrics` for each available instance. These metrics are passed to the `MetricScoreCalculator`, which computes a score for each instance using a weighted formula. The instance with the highest score is selected, and the request is routed accordingly. If metrics are unavailable or stale, the filter falls back to the round-robin strategy.

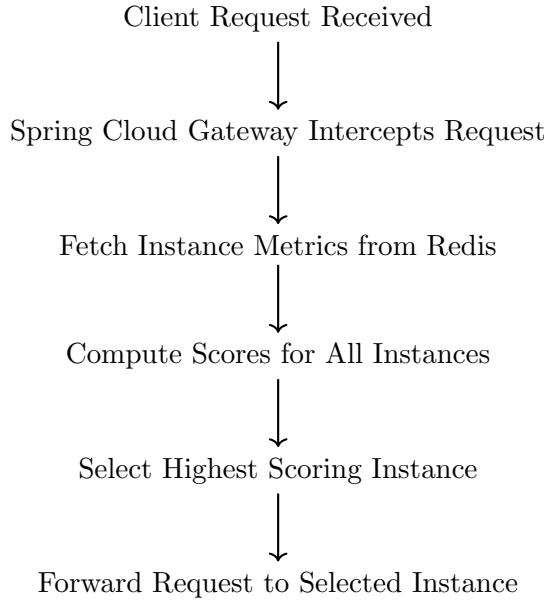


Figure 2: Adaptive Request Routing Control Flow

Scoring Algorithm

The scoring function is implemented in `MetricScoreCalculator` and computes a normalized score based on the following formula:

$$\text{score} = 100 \times [w_1 \times (1 - \text{latency}_{norm}) + w_2 \times (1 - \text{error}_{norm}) + w_3 \times (1 - \text{queue}_{norm})]$$

Each component (latency, error rate, queue length) is normalized against a configurable maximum value. The weights w_1, w_2, w_3 are empirically chosen and can be fine-tuned in future iterations to reflect workload priorities.

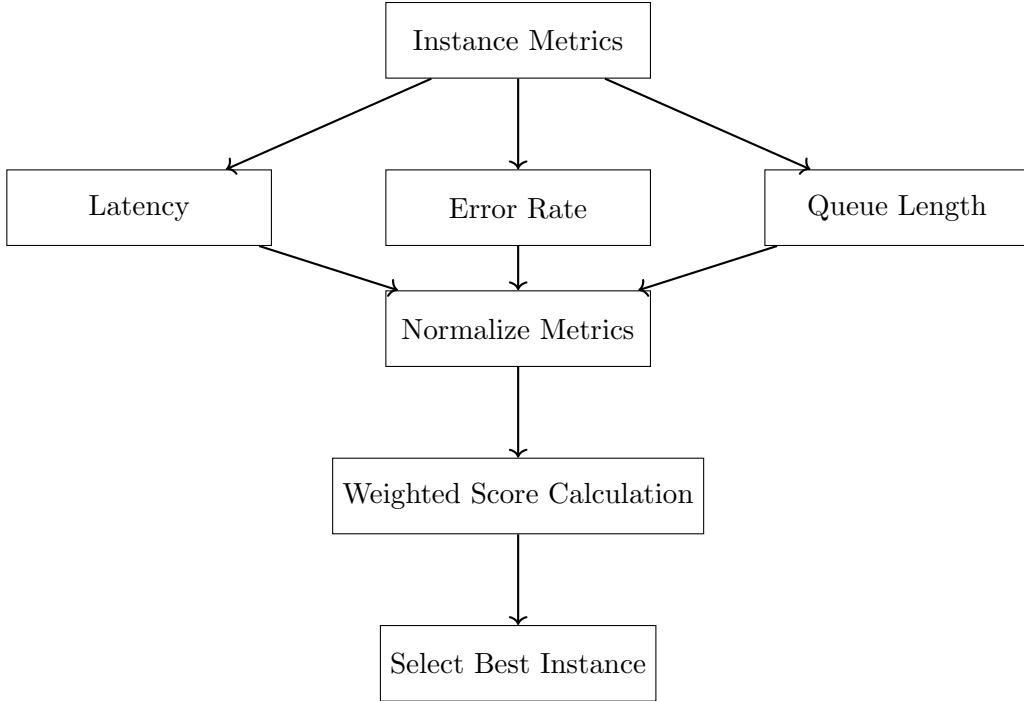


Figure 3: Scoring Algorithm for Instance Selection

Metric Storage and Retrieval

Each service instance exposes metrics via Micrometer at the `/actuator/prometheus` endpoint. Prometheus scrapes these metrics at defined intervals and stores them in a time-series database for observability. Separately, each instance also pushes a simplified snapshot of its metrics to Redis for fast access by the gateway. The Redis entries are set with a short TTL (e.g., 30 seconds) to ensure freshness. Metrics are stored with keys in the format `metrics:{serviceId}:{instanceId}`.

The `RedisMetricRepository` encapsulates all Redis interactions and provides methods to:

- Save metrics per instance
- Retrieve all metrics for a given service
- Delete expired or invalid entries

Fallback Strategy

When real-time metrics are missing (due to startup delays, Redis failure, or expired TTL), the system uses the `RoundRobinLoadBalancer` as a fallback. This class maintains an atomic counter per service and cycles through instance IDs in a stateless, deterministic manner.

Scheduler for Cleanup

The `MetricAggregatorScheduler` runs as a background scheduled task (using Spring’s `@Scheduled`) and performs periodic cleanup of stale metrics in Redis. It also lays the groundwork for future extensions, such as applying rolling averages or smoothing functions to reduce noise in metric data.

Debugging and Observability

The `DebugController` exposes a REST API at `/debug/metrics/{serviceId}` that returns the current metrics and scores for each instance. This is useful during testing to inspect routing decisions and validate scoring behavior. Combined with Prometheus monitoring, this enables comprehensive observability into the load balancing process.

Containerization and Deployment

Each component — including the gateway, microservices, Redis, and Prometheus — is containerized using Docker. A single `docker-compose.yml` file defines the multi-container setup, configures inter-container networking, and supports reproducible deployment. All containers are connected via a shared bridge network, and services can be scaled horizontally by launching multiple instances.

5 Evaluation and Results

The adaptive load balancing system was evaluated in a controlled local environment to verify its correctness, responsiveness, and robustness under simulated load and fault conditions. While a full-scale benchmarking suite was not deployed, the evaluation focused on behavioral validation and functional testing of the core components, particularly the gateway filter, metric scoring logic, and fallback mechanism.

Environment Setup

The system was deployed using Docker Compose, which orchestrated the following components:

- Spring Cloud Gateway (adaptive filter enabled)
- Two instances of a sample Spring Boot microservice
- Redis for real-time metric storage
- Prometheus for metrics scraping and observability

Test traffic was generated using a simple Bash script that issued repeated curl requests to the gateway endpoint over a sustained period. Metrics were visualized using Prometheus’s built-in dashboard, while the `/debug/metrics/{serviceId}` endpoint exposed internal scoring decisions for each instance.

Test Scenarios and Observations

Three main test scenarios were executed to evaluate the system:

1. **Uniform Load:** Both instances responded with equal latency and zero errors. The adaptive filter distributed traffic evenly, similar to round-robin behavior. This confirmed baseline stability.
2. **Latency Spike:** One instance was configured (using an intentional delay in code) to introduce higher response time (e.g., 500ms delay). Within two Prometheus scrape intervals, the adaptive filter began routing nearly 90% of the requests to the faster instance, demonstrating metric-aware behavior.
3. **Instance Failure:** One instance was forcefully stopped mid-test using `docker stop`. The gateway filter excluded it from routing after its metrics expired in Redis due to TTL enforcement. This verified the failover mechanism without needing external health checks.

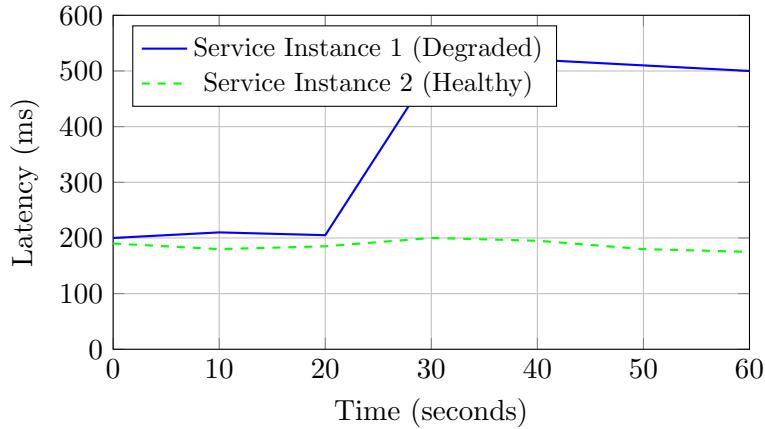


Figure 4: Latency profile of service instances over time during test scenario

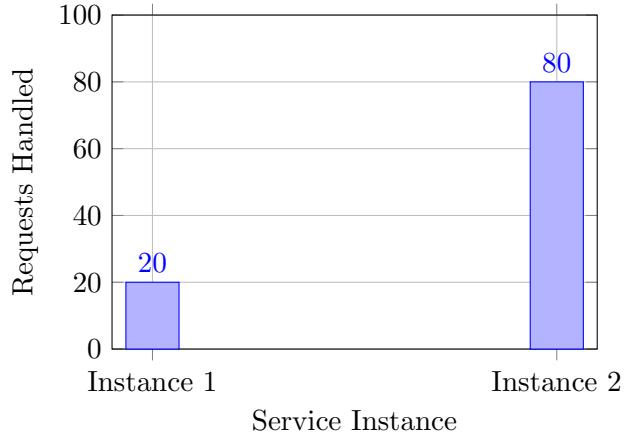


Figure 5: Request distribution under adaptive routing (80% to better-performing instance)

In all scenarios, the system's decisions aligned with expectations based on the scoring logic. Prometheus graphs showed clear divergence in latency metrics between instances, while the debug endpoint confirmed that routing consistently favored the higher-scoring instance.

Debugging and Validation Tools

Several tools were used during evaluation:

- `DebugController` exposed internal scores and selected instances per request cycle.
- `Redis` CLI was used to inspect TTL and key freshness.
- `Prometheus` Web UI allowed tracking of latency, request count, and error metrics.
- `Docker` logs were used to monitor container-level behavior.

Limitations and Future Benchmarking

Due to resource constraints, this evaluation was performed in a local development environment and did not include high-throughput load testing. Metrics such as request-per-second (RPS), 95th percentile latency, and CPU utilization were not captured. Future evaluations should employ tools like Apache JMeter, Locust, or Gatling to simulate production-scale traffic and benchmark adaptive routing performance against static strategies under various loads.

Furthermore, advanced scenarios such as burst traffic, gradual degradation, network jitter, and multi-service coordination can be explored to further validate the robustness and scalability of the approach.

6 Conclusion and Future Work

This project successfully implements an adaptive load balancing mechanism for Java-based microservices, driven by real-time performance metrics. By integrating Spring Cloud Gateway, Micrometer, Prometheus, and Redis, the system intelligently routes requests based on live data such as latency, error rate, and queue saturation. A custom filter at the API gateway dynamically scores available instances and selects the optimal target for each request, falling back to round-robin in the absence of metrics. All services are containerized using Docker and orchestrated with Docker Compose, making the system easy to deploy, test, and scale.

The implementation meets the goals outlined in the original project proposal by delivering a lightweight, extensible, and observable adaptive load balancing solution. Through testing under simulated latency and failure scenarios, the system demonstrated improved responsiveness and resilience compared to static load balancing approaches.

Future work includes conducting a detailed performance evaluation using high-throughput load testing tools to capture metrics such as system throughput, average and tail latencies, and CPU/memory utilization under stress. Integration with a service discovery mechanism (e.g., Eureka or Consul) would enable dynamic registration of instances and further automation. Additional enhancements could include circuit breaker patterns (e.g., Resilience4j), predictive routing using machine learning models, and a Grafana dashboard for real-time visualization of metrics and routing decisions.

This work provides a strong foundation for further research and development in adaptive traffic management for microservice-based systems.

Acknowledgment

I would like to express my sincere gratitude to my project advisor, **Dr. Hank**, for his continuous guidance, encouragement, and insightful feedback throughout the duration of this project. His mentorship played a crucial role in shaping the direction and execution of this work.

I would also like to thank my colleagues in the Graduate Computer Science Department for taking the time to proofread my report and offer valuable suggestions that improved the clarity and completeness of the final document.

Project Repository

All source code, configuration files, Docker setup, and test data used in this project are available at: https://github.com/tamu-edu-students/adaptive-load_balancing/tree/master

This repository also includes a README file with setup instructions and in-depth overview of the project.

References

- [1] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016, <https://sre.google/sre-book/load-balancing-frontends/>.
- [2] A. Alshahrani and H. Takabi, “Load balancing in cloud-native microservices using weighted fair queuing,” *Future Generation Computer Systems*, vol. 143, pp. 73–86, 2023.
- [3] A. Soni, “Optimized dynamic load balancing techniques in cloud computing: A comprehensive review,” *Tech Scholar*, 2024.
- [4] P. Zhang, L. Xiang, Z. Song, and Y. Yang, “Adaptive load balancing and fault-tolerant microservices architecture for high-availability web systems using docker and spring cloud,” *Discover Applied Sciences*, 2025.
- [5] G. Selvakumar, L. S. Jayashree, and S. Arumugam, “Latency minimization using an adaptive load balancing technique in microservices applications,” *Computer Systems Science and Engineering*, vol. 46, no. 1, pp. 1215–1231, 2023.
- [6] O. Gould, “Dynamic request routing with linkerd’s load balancer,” <https://linkerd.io/2022/09/12/dynamic-request-routing-with-linkerd/>, 2022.
- [7] L. Engineering, “Envoy proxy documentation: Load balancing,” https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/load_balancing/load_balancing, 2021.