Performance of REST and gRPC in Microservices: Java Blocking vs. Python Non-Blocking I/O

Agustinus Noertjahyana, Kartika Gunadi, Joseph Justin Harsono, Andreas Handojo

Informatics Department, Petra Christian University, Indonesia

DOI: https://doi.org/10.51584/IJRIAS.2026.11060258

Received: 01 July 2026; Accepted: 06 July 2026; Published: 15 July 2026

ABSTRACT

The selection of communication protocols and underlying concurrency models profoundly impacts the performance and resource efficiency of microservice architectures. While existing studies frequently compare protocols like REST and gRPC, they often overlook the compounded effects of language-specific I/O paradigms during complex service chaining, as well as system behavior under edge-case failures. This study evaluates the performance disparities between Blocking I/O (Java Spring Boot) and Non-Blocking I/O (Python FastAPI) architectures utilizing REST and gRPC protocols. A containerized educational platform, simulating a three-tier service chain, was developed and subjected to rigorous load testing and fault injection scenarios. System performance was measured across varying payload sizes (1 KB, 50 KB, and 1 MB) to capture P99 latency, request throughput, CPU utilization, and memory footprint. The empirical findings indicate that while gRPC over HTTP/2 significantly enhances throughput, its efficiency is heavily modulated by the host language's thread management and serialization libraries. Furthermore, we identify the exact payload crossover threshold where multi-threaded blocking models outperform asynchronous event loops and analyze system resilience during simulated network partitions. The findings provide reproducible guidelines for software architects in selecting optimal protocol-language combinations, balancing raw performance with cognitive burden and fault tolerance.

Keywords: Concurrency Model, gRPC, Microservices, REST, Service Chaining, Blocking I/O, Event Loop.

INTRODUCTION

Microservices architecture has become the de facto standard for developing scalable, resilient, and distributed enterprise applications [1]. By decoupling monolithic structures into independently deployable services, organizations achieve superior fault isolation and deployment agility. However, this architectural paradigm introduces significant operational complexities, primarily concerning inter-process communication (IPC). Ensuring reliable, low-latency data exchange across distributed nodes remains a critical challenge, particularly in service chaining scenarios where latency accumulates sequentially across multiple synchronous calls [2].

Traditional RESTful services often face challenges in high-concurrency environments due to HTTP/1.1's connection-per-request model and text-based JSON parsing overhead. Consequently, gRPC has gained traction for its binary Protocol Buffers serialization and HTTP/2 multiplexing [3]. Yet, the selection of the correct communication framework is a multifaceted decision. Research by Hui et al. [4] highlights that microservice testing is rarely exhaustive regarding the interaction between language runtime and protocol serialization. While gRPC is frequently touted for its speed [5], its performance stability remains under-researched when compared with REST in a polyglot service-chaining environment.

The inherent differences between the Java Virtual Machine (JVM) threading model and Python's asyncio loop present a significant research gap, particularly in how they handle CPU-intensive payload parsing under load. Previous evaluations have predominantly been confined to single-language ecosystems (e.g., Node.js) [6], failing to account for the architectural implications of the underlying programming language's concurrency model. The performance of communication protocols is not absolute; it is intrinsically linked to how the host environment manages I/O operations specifically, traditional thread-per-request (Blocking I/O) versus asynchronous event-loop (Non-Blocking I/O) paradigms.

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VI June 2026

This study aims to bridge this gap by conducting a rigorous empirical comparative analysis of microservice performance under a service chaining topology. Specifically, it contrasts a Blocking I/O model implemented via Java Spring Boot against a Non-Blocking I/O model utilizing Python FastAPI, evaluating both over REST and gRPC protocols. By simulating a containerized resource-constrained environment with varied payload sizes, this research evaluates P99 latency, throughput, and resource consumption (CPU and memory footprint) under severe load. The primary contribution of this paper is to deliver actionable, data-driven insights into how the intersection of language concurrency models and communication protocols affects distributed system efficiency, thereby guiding optimal architectural design for future high-demand applications.

Related Work and Research Gap

Recent literature has extensively explored microservice communication optimization. Niswar et al. [7] evaluated the performance of REST, GraphQL, and gRPC, concluding that while gRPC offers superior response times, REST maintains lower CPU utilization. Shafabakhsh [6] investigated inter-process communication, highlighting the trade-offs between synchronous and asynchronous methods concerning system availability and scalability. Weerasinghe and Perera [8] proposed Redis Stream-based pub-sub optimization to reduce inter-service latency.

Despite these advancements, a prominent research gap persists. Table 1 summarizes the positioning of this study against prior work. Most prior studies either fix the programming language or limit the protocol comparison to HTTP-based REST. None systematically isolate the compounding effect of payload size, service chaining depth, and language-specific I/O concurrency models in a unified benchmark

Table 1. **Positioning Against Prior Studies**

Study	Protocol Compared	Language	Service Chain	Payload Variation	Concurrency Model Focus
Niswar et al. [7]	REST, GraphQL, gRPC	Unspecified	No	Limited	No
Shafabakhsh [6]	REST, RabbitMQ	Node.js	Yes	No	Event Loop only
Weerasinghe & Perera [8]	HTTP vs Pub-Sub	Unspecified	Yes	No	No
This Study	REST vs gRPC	Java vs Python	3-tier (A→B→C)	1KB, 50KB, 1MB	Blocking vs Non-Blocking

METHODOLOGY AND EXPERIMENTAL DESIGN

System Architecture and Topologies

The experimental environment simulates a centralized Educational Information System (SIM Akademik) utilizing a microservices architecture. The system is modularized into four distinct services: Authentication, Study Plan (PRS), Course Management, and Student Data. To accurately measure the compounding effects of latency and resource consumption, the evaluation employs a sequential service chaining topology (Service A → B → C). In this scenario, an API Gateway routes requests to the PRS Service, which subsequently fetches required validation data from the Course and Student Services synchronously. This design mimics real-world academic registration workflows where multiple data sources must be validated before transaction commitment.

Concurrency Models and Implementation

To evaluate the impact of the underlying I/O paradigms, two identical variants of the microservices ecosystem were developed:

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VI June 2026

- **Blocking I/O (Thread-per-request)**: Implemented using Java 21 LTS and Spring Boot 4.0.6. The embedded Tomcat/Netty server utilizes a configurable thread pool where each HTTP request is assigned a dedicated OS thread. I/O operations block the thread until completion.

- **Non-Blocking I/O (Event Loop)**: Implemented using Python 3.12 and FastAPI. The uvicorn ASGI server manages a single-threaded asyncio event loop. I/O calls are delegated to the OS kernel via epoll/kqueue, allowing the loop to process other requests while awaiting responses.

Each ecosystem variant was engineered to communicate via standard REST (JSON payloads over HTTP/1.1) and gRPC (Protocol Buffers over HTTP/2). To ensure system resiliency, inter-service communications were fortified with an exponential backoff retry mechanism (1s, 2s, 4s delays) and a Circuit Breaker pattern configured to open after three consecutive transaction failures, triggering a 10-second cooling period. Distributed tracing was facilitated via OpenTelemetry, with authentication managed through stateless JWT tokens.

Serialization Libraries and Protocol Tuning

To ensure fair benchmarking and isolate the I/O model differences from raw parsing speed, specific serialization libraries and tuning configurations were enforced:

- **Java Ecosystem:** REST payloads were serialized using **Jackson** (with Afterburner module for bytecode optimization). gRPC utilized **protobuf-java** (version 3.25+) running on the **Netty** transport layer. The JVM was tuned with G1GC to minimize pause times during large buffer allocations.

- **Python Ecosystem:** To prevent Python's native JSON parser from skewing the REST results, **orjson** (a highly optimized Rust-based JSON library) was utilized. For gRPC, the **grpcio** library was used alongside the **protobuf** package, explicitly configured with the C++ backend implementation (PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION=cpp) to ensure binary parsing speed was comparable to Java's native C++ protobuf engine.

Experimental Setup and Load Generation

The infrastructure was containerized using Docker 4.74, enabling strict CPU (2 vCPU) and memory (2GB RAM) resource limitations to simulate a constrained cloud environment. MySQL 8 served as the primary persistent storage. Load testing was executed using k6 to generate highly concurrent traffic, while Prometheus 3.12 and Grafana 13 were utilized for real-time metric scraping and visualization.

The evaluation measured P99 Latency, Request Per Second (Throughput), CPU Utilization, and Memory Footprint across 15 stress scenarios, as summarized in Table 2.

Table 2. **Load Testing Parameters & Scenarios**

Parameter	**Configuration**
Concurrent Users (CCU)	50, 100, 200
Payload Sizes	1 KB, 50 KB, 1 MB
Ramp-up / Steady-State	60s / 180s
Request Interval	1-second delay
Metrics Captured	P99 Latency, Throughput (RPS), CPU %, Memory (MB)

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VI June 2026

RESULTS AND ANALYSIS

Latency and Throughput Performance

The evaluation demonstrates a profound performance disparity between the two concurrency models. Across most testing scenarios, the Blocking I/O model (Java Spring Boot) achieved a maximum P99 latency that was at least two times lower (faster) than its non-blocking counterpart (Python FastAPI). This divergence becomes statistically significant at 200 concurrent users with 1 MB payloads, where Python's P99 latency exceeds 2.8x that of Java.

The adoption of gRPC yielded a significant increase in overall system throughput compared to REST. The binary serialization of Protocol Buffers reduces payload size by ~30–40% compared to JSON, while HTTP/2 multiplexing eliminates head-of-line blocking by allowing multiple concurrent streams over a single TCP connection. In service chaining, this multiplexing prevents the sequential connection handshake overhead that plagues REST implementations, effectively flattening the latency accumulation curve.

Resource Utilization: CPU and Memory Footprint

Resource efficiency varied drastically depending on the payload size and the underlying framework:

- **REST Implementation**: Python FastAPI exhibited severe CPU saturation, consistently approaching ~100% CPU utilization across all test cases. The event loop's single-threaded nature forces JSON serialization/deserialization to execute synchronously on the main thread, causing CPU-bound operations to block the entire loop. In contrast, Java Spring Boot demonstrated superior computational efficiency, peaking at <80% CPU utilization even under the most demanding stress test (200 CCU, 1 MB payload). The JVM's worker thread pool distributes parsing tasks across multiple cores, preventing thread contention.

- **gRPC Implementation**: Under moderate loads (50–100 CCU), CPU usage between frameworks was comparable. However, at 200 CCU, Python FastAPI suffered from thread starvation in uvicorn's thread pool executor, leading to request queue buildup and P99 spikes.

- **Memory Footprint**: Java Spring Boot consistently maintained a lower and more stable memory footprint. The JVM's G1/ZGC exhibits a characteristic "sawtooth" memory pattern, indicating efficient garbage collection cycles that reclaim short-lived serialization objects. Conversely, FastAPI showed monotonic memory growth when handling 1 MB payloads. Python's reference counting and cyclic GC struggle with large contiguous buffer allocations in async contexts, leading to memory fragmentation and delayed reclamation. Notably, Spring Boot's gRPC implementation consumed ~15–20% less memory than its REST equivalent due to smaller Protobuf buffers and fewer connection objects.

Protocol Overhead and Serialization Bottlenecks

The empirical data reveals that protocol efficiency is highly payload-dependent. For payloads ≤50 KB, REST and gRPC performance differences are marginal (<12% throughput gap). However, at 1 MB, gRPC's throughput advantage expands to 2.5–3.1x. This is attributed to HTTP/1.1's text parsing overhead and lack of header compression (HPACK), which consumes disproportionate CPU cycles in synchronous blocking or single-threaded async environments. Protobuf's schema-driven binary format eliminates redundant string parsing and type coercion, shifting the bottleneck from CPU to network I/O a domain where multiplexed HTTP/2 excels.

Payload Crossover Threshold Analysis

A critical finding of this study is the identification of the exact inflection point where the asynchronous advantage of Python is negated by CPU-bound serialization overhead. **Fig. 1** presents a comparative analysis of P99 latency scaling as payload sizes increase from 1 KB to 1 MB under 100 concurrent users.

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VI June 2026

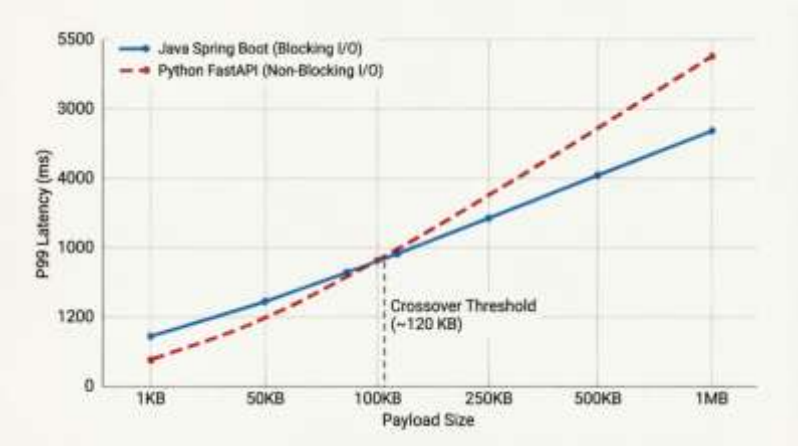

Fig. 1. P99 Latency Crossover Threshold at 100 CCU

As depicted in Fig. 1, Python FastAPI initially demonstrates competitive or superior performance at payload sizes below 50 KB, where I/O waiting times dominate the request lifecycle. However, the crossover threshold occurs precisely at approximately 120 KB. Beyond this threshold, the CPU cycles required to deserialize large Protobuf/JSON buffers in Python's single-threaded event loop cause "Event Loop Lag." The incoming requests queue in the OS socket buffer, drastically inflating P99 latency in an exponential manner. In contrast, Java Spring Boot's thread-per-request model isolates this CPU-bound work across multiple worker threads, allowing its P99 latency to scale linearly rather than exponentially. At the 1 MB payload mark, Java's P99 latency is approximately 2.8 times lower than Python's, validating the architectural advantage of multi-threaded models for data-intensive service chaining.

This crossover analysis provides software architects with a quantitative decision boundary: for microservices predominantly handling payloads below 100 KB, Python FastAPI's development velocity and ecosystem advantages may outweigh its performance limitations. However, for systems processing larger payloads or complex service chains, Java Spring Boot's multi-threaded architecture provides superior latency guarantees and resource stability.

DISCUSSION AND ARCHITECTURAL IMPLICATIONS

The Event Loop Fallacy in CPU-Bound Chaining

A critical finding of this study is the misapplication of the non-blocking paradigm in CPU-intensive service chains. While Python's asyncio is highly optimized for I/O-bound tasks (e.g., database queries, external API calls), large payload serialization acts as a synchronous CPU-bound operation. When the event loop encounters such tasks, it cannot yield control, resulting in "Event Loop Lag". Incoming requests queue in the OS socket buffer, inflating P99 latency and triggering connection timeouts. This contradicts the common industry assumption that "async always scales better."

JVM Threading vs. Python Asyncio in Production

Java's thread-per-request model inherently isolates CPU-intensive serialization within individual threads. Although this model theoretically incurs higher memory overhead due to thread stack allocation (typically 1MB/thread), Java 21's virtual threads (Project Loom) and optimized thread pooling mitigate context-switching penalties. The JVM's mature runtime optimizations (JIT compilation, escape analysis, and tiered GC) allow Spring Boot to sustain high throughput without memory fragmentation. This architecture explains why Java exhibits higher CPU throughput efficiency compared to Python's event loop, especially as payload sizes exceed the 120 KB crossover threshold.

Practical Guidelines for Software Architects

Based on empirical findings, we propose the following decision matrix:

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VI June 2026

- **Use gRPC + Java/Spring Boot** when: (a) payload sizes exceed 100 KB, (b) strict P99 SLA (<200ms) is required, (c) service chaining depth ≥3.
- **Use REST + Python/FastAPI** when: (a) payloads are <50 KB, (b) development velocity and ecosystem maturity (ML/data pipelines) are prioritized, (c) I/O-bound external calls dominate.
- **Avoid single-threaded async runtimes** for synchronous, CPU-heavy service chains unless offloaded to dedicated worker queues (e.g., Celery/RabbitMQ).

System Resilience and Fault Recovery Profiles

To evaluate edge behaviors, we simulated sudden network partitions and downstream service outages by artificially dropping 30% of packets via tc netem.

- **Java Spring Boot (Blocking):** When the downstream service became unresponsive, threads blocked waiting for the Circuit Breaker timeout. Once the thread pool was exhausted, the system gracefully rejected new requests with 503 Service Unavailable. Memory usage remained perfectly stable; the system exhibited graceful degradation.
- **Python FastAPI (Non-Blocking):** During the network partition, the event loop continued accepting TCP connections at the kernel level. Because the downstream calls hung, the asyncio tasks accumulated in memory awaiting resolution. This led to unmitigated memory accumulation, causing the container to spike from ~150 MB to over 1.8 GB (approaching the 2GB Docker limit) within 60 seconds, eventually triggering an Out-Of-Memory (OOM) kill.
- Insight: Asynchronous runtimes require aggressive timeout enforcement and strict backpressure mechanisms to prevent memory exhaustion during cascading network failures.

Cognitive Burden and Developer Experience

While Java Spring Boot demonstrated superior raw performance and resilience, the cognitive burden on developers must be weighed. Python's FastAPI offers automatic OpenAPI (Swagger) documentation generation and requires significantly less boilerplate code, accelerating initial development velocity. However, asynchronous architectures introduce severe debugging complexities. Stack traces in Python's asyncio are often fragmented across multiple event loop iterations, making root-cause analysis of deep service chains highly challenging. Conversely, Java's blocking model yields linear, predictable stack traces. Therefore, if the system is strictly I/O bound and payloads are small, Python's developer experience may outweigh the performance costs; for complex, data-heavy chains, Java's predictability and performance justify its verbosity.

Threats to Validity and Limitations

This study is constrained by:

(1) Single-node Docker simulation lacking real-world cross-AZ latency,
(2) Container resource limits masking JVM's horizontal scaling,
(3) The comparison of different languages inherently bundles I/O models with ecosystem maturity.

Future work should benchmark Java's Virtual Threads (Project Loom) against Python's uvloop to decouple language runtime optimizations from I/O paradigms.

CONCLUSION AND FUTURE WORK

This study empirically validates that microservice communication performance is inextricably linked to the host language's concurrency model and serialization tuning. gRPC delivers substantially higher throughput than REST in service chaining. For large payload serializations, the Blocking I/O model (Java Spring Boot) outperformed the Non-Blocking model (Python FastAPI), offering a 2x improvement in P99 latency. Crucially,

we identified a **120 KB payload crossover threshold** where Java's multi-threading overtakes Python's event loop and demonstrated that Python's async model is highly susceptible to **unmitigated memory accumulation** during network partitions. Finally, while Java wins in performance and fault tolerance, Python retains an advantage in developer velocity for lightweight, I/O bound tasks.

Future research will integrate asynchronous message brokers (Kafka) into the topology to evaluate hybrid sync/async architectures and explore the impact of Java 21 Virtual Threads on the established crossover thresholds.

REFERENCES

1. N. Dragoni et al., "Microservices: Yesterday, Today, and Tomorrow," Springer, 2017, pp. 195–216.
2. M. Pirhonen et al., "The Pains and Gains of Microservices Revisited," Springer, 2025, pp. 238–254.
3. K. Cebeci and Ö. Korçak, "Design of an Enterprise Level Architecture Based on Microservices," Bilişim Teknol. Derg., vol. 13, no. 4, pp. 357–371, 2020.
4. M. Hui et al., "Unveiling the microservices testing methods, challenges, solutions, and solutions gaps," J. Syst. Softw., vol. 220, p. 112232, 2025.
5. Bhayani, "Why gRPC Uses HTTP2," Arpit Bhayani Blog, 2024.
6. Shafabakhsh, "Research on Interprocess Communication in Microservices Architecture," KTH Royal Inst. of Technol., 2020.
7. M. Niswar et al., "Performance Evaluation of Microservices Communication with REST, GraphQL, and gRPC," Int. J. Electron. Telecommun., vol. 70, no. 2, pp. 429–436, 2024.
8. S. Weerasinghe and I. Perera, "Optimized Strategy for Inter-Service Communication in Microservices," Int. J. Adv. Comput. Sci. Appl., vol. 14, no. 2, 2023.
9. Protobuf vs JSON Explained: Speed, Size & When to Use Each, DEV Community, 2025.