E-Commerce Flash Sale System

Performance Analysis Report

Written By:

- Bayan AboAlrob
- Duaa Braik

Contents

1. Introduction

Performance evaluation plays a central role in the development of systems intended for high-traffic environments such as flash sales. These platforms frequently experience sudden, unpredictable spikes in traffic, where thousands of users attempt to purchase a limited number of items at the same moment. Ensuring fairness, consistency, and responsiveness during these critical load periods requires not only a robust architectural design but also a clear understanding of how the system behaves under real stress conditions.

This report presents a comprehensive performance analysis of our E-Commerce Flash Sale System, built using two distinct architectural approaches:
- a **Synchronous HTTP-based approach**, and
- an **Asynchronous Event-Driven approach using RabbitMQ**.

The objective of this analysis is to evaluate how each architecture handles extreme concurrency, how load affects latency and throughput, and how system resources such as CPU and memory respond to simultaneous interactions. In addition, this report examines qualitative factors—maintainability, complexity, scalability constraints, and operational behavior—to build a complete picture of the strengths and limitations of each design.

2. Performance Testing Methodology

This section explains the tools, infrastructure, execution environment, and testing strategies used to measure the performance of both architectural approaches in a controlled and repeatable manner.

2.1 Testing Tools

To ensure realistic and reliable measurements, we used a combination of load-testing tools and performance-monitoring systems. The primary load generator was **k6**, chosen for its ability to simulate high concurrency and its clear, developer-friendly scripting model. Using k6, we modeled the complete end-to-end checkout scenario—login, cart creation, product fetch, item addition, reservation creation, and payment confirmation—to reproduce the real user journey during a flash sale.

However, k6 alone cannot measure server-side CPU or memory usage. For this reason, we integrated **Grafana** and **InfluxDB** to capture live operational metrics for each microservice. A custom background metrics collector was added to every service to push periodic CPU and memory data into an InfluxDB bucket. Grafana was then used to visualize this data in real time, enabling us to observe how each architecture behaved internally under increasing load.

2.2 Metrics Infrastructure (Grafana + InfluxDB)

To build a stable and consistent monitoring environment, we deployed InfluxDB using a Docker container and connected Grafana to it as a data source.

InfluxDB was installed using:

docker run -d --name influxdb -p 8086:8086 influxdb:2.7

Within InfluxDB, we created a dedicated organization, bucket, and authentication token. These credentials were then used by every microservice's background metrics collector to push CPU and memory measurements.

Grafana (running locally on port 3000) was connected to the InfluxDB instance. After binding the data source, we constructed a dashboard that aggregated metrics across all services. This included panels showing:

-CPU usage per microservice

-Memory consumption per microservice

-Summed CPU usage across the entire system

-Summed memory utilization

-Latency trends and throughput visualizations

2.3 Test Environment and Hardware Specifications

All load tests were executed on a local machine with consistent hardware to eliminate variability. The specifications were as follows:

Device: ASUS

CPU: Intel Core i7 with 8 physical cores, 16 threads (logical processors)

Memory: 16 GB RAM

Operating System: Windows 10 PRO, version 10.0.19045

2.4 Test Scenarios

We evaluated two architectural approaches under the same load conditions:

1. Synchronous (HTTP Request–Response):
In this scenario, services communicated sequentially using direct HTTP calls. Nginx applied a **Round-Robin** load-balancing strategy, distributing incoming requests evenly across multiple service instances. This reflected a traditional microservices environment where each action depends on the successful completion of the previous one.

2. Asynchronous (Event-Driven with RabbitMQ):
In this approach, the Reservation, Order, Payment, and Inventory services communicated through events published to RabbitMQ. Nginx applied a **Least Connection** strategy to ensure that services receiving direct traffic were routed optimally. This model reflects a loosely coupled, non-blocking architecture where independent services react to events rather than wait for responses.

Both scenarios were tested with **500 and 250 Virtual Users (VUs)** executing the same logical workflow, ensuring a fair comparison.

2.5 Test Duration and Repetitions

To minimize anomalies and guarantee statistical reliability, each test scenario was executed multiple times. The testing phase consisted of:

-**Two warm-up runs** using 250 virtual users

-**Three full-scale runs** using 500 virtual users

This resulted in **five execution cycles** per architecture, allowing us to average the results and analyze both typical and edge-case behaviors.

3. Quantitative Analysis
This section presents the numerical findings from the k6 load tests and the system-level metrics collected from Grafana and InfluxDB.

3.1 Summary of Raw k6 Results

The asynchronous architecture demonstrated significantly faster average request processing due to its non-blocking nature. However, it experienced a higher failure rate due to message retries and RabbitMQ timeouts under peak load. Conversely, the synchronous architecture maintained a lower failure rate but suffered severe latency penalties as concurrency increased.

3.2 Comparison Table

Results for 250 virtual users:

Metric	**Synchronous+Round Robin**	**Asynchronous+ Least connections**	**Winner**
Avg Latency	1.6s	753.49ms	**Async**
P90 Latency	3.43s	1.22s	**Async**
P95 Latency	4.13s	1.54s	**Async**
CPU Usage	2.5%	3.56%	**Sync**
Memory Usage	163.72 MB	363 MB	**Sync**
Failure Rate	0.00%	4.48%	**Sync**
Throughput(req/s)	114.34	153.20	**Async**

Results for 500 virtual users:

Metric	**Synchronous+Round Robin**	**Asynchronous+ Least connections**	**Winner**
Avg Latency	2.22s	1.2s	**Async**
P90 Latency	4.97s	2.06s	**Async**
P95 Latency	6.86 s	2.42s	**Async**
CPU Usage	3.2%	5%	**Async**
Memory Usage	133.36 MB	166.82 MB	**Sync**
Failure Rate	13.27%	14.32%	**Sync**
Throughput(req/sec)	107.49	136.71	**Async**

These results highlight a fundamental architectural trade-off:

-synchronous design completes more requests overall, but with extremely high latency.

-asynchronous design responds much faster but experiences more failures under load.

4. Qualitative Analysis

4.1 Situations Where the Synchronous Approach Is Preferable

The synchronous architecture is most suitable for systems requiring **strict, immediate consistency**. Because each operation waits for the previous one to complete, the workflow remains predictable, linear, and easy to reason about. This model is ideal in scenarios where the order of operations matters significantly and where the infrastructure can confidently maintain low to moderate concurrency levels.

However, this design inherently sacrifices performance under heavy load because blocking requests accumulate and amplify latency. Therefore, synchronous systems work best when reliability and consistency are more important than scalability or responsiveness.

4.2 Situations Where the Asynchronous Architecture Is Preferable

The asynchronous approach is better suited for dynamic, large-scale environments where high concurrency is expected. It decouples services through events, enabling each component to operate independently without blocking downstream workflows. This architecture thrives under heavy user traffic, ensures better responsiveness, and naturally supports horizontal scaling.

The trade-off is the absence of immediate consistency, as actions may propagate through the system over time. Asynchronous systems introduce greater complexity in terms of tracing, debugging, and ensuring idempotent behavior—but gain immense flexibility and robustness under real-world load.

4.3 Scalability Limitations

Both architectures face specific scalability boundaries:

-Synchronous scalability limits:
Thread pools are finite, and as concurrency grows, response times degrade dramatically. The system becomes prone to cascading failures if a single service slows down.

-Asynchronous scalability limits:
Message queues may become bottlenecks, and large bursts of events can cause delays or retries. More components also mean higher operational overhead.

4.4 Real-world Considerations (Cost, Maintenance, Complexity)

From a real deployment perspective, cost, complexity, and maintainability become essential factors:

-Synchronous systems are cheaper to host and simpler to maintain but cannot handle flash-sale level traffic.

-Asynchronous systems provide scalability but require message brokers, additional monitoring, and stronger operational practices.

4.5 Key Observations & Unexpected Results

Several observations emerged during testing:

-Synchronous workflows slowed far earlier than expected, revealing sensitivity to even moderate load.

-Asynchronous workflows maintained fast responses but encountered higher failure rates due to retries, high cache overload and queue pressure.

5. Lessons Learned

5.1 Issues Faced & How They Were Solved

Two critical issues affected the performance results:

1.Database Connection Exhaustion:
Under load, database locks caused connection pooling failures. We resolved this by increasing the maximum pool size to **200**, which stabilized the synchronous flow significantly.

2.RabbitMQ Queue Consumption Behavior:
Initially, all services consumed events from a **single queue**, resulting in only one subscriber receiving the message. We fixed this by creating a **dedicated queue per service**, ensuring proper multi-consumer behavior.

3.Redis timeouts appeared only in the asynchronous approach

because async processing generated far more parallel operations on the same Redis instance. Unlike the synchronous flow—which naturally serialized traffic—the async

model allowed multiple services and consumers to hit Redis at the same time, creating hot-key pressure and filling Redis' command and connection queues. This overload led to client-side timeouts and revealed that shared resources must be scaled or tuned differently in high-concurrency async architectures.

Additionally, since k6 lacks resource-level monitoring, we introduced Grafana + InfluxDB collectors to fill this gap.

5.2 What We Would Do Differently

If we were to repeat this project in a real production environment, we would:

- ✧ Containerize every service using Docker
- ✧ Deploy on Kubernetes with horizontal autoscaling
- ✧ Use Prometheus and OpenTelemetry for deeper observability
- ✧ Run tests on cloud infrastructure instead of a local machine
- ✧ Integrate circuit breakers, rate limiting, and bulkheading for resilience

5.3 Application of Course Concepts

This project successfully applied numerous architectural principles taught during the course, such as:

- Designing a fully event-driven microservice system.

-Implementing synchronous request–response workflows .

-Applying Redis caching strategies to reduce pressure on SQL databases.

-Using API Gateway patterns for routing and load balancing

-Understanding that architecture involves trade-offs, not absolute answers.

5.4 Future Improvements

While the current implementation successfully demonstrates both synchronous and asynchronous architectures under realistic flash-sale load, there is still significant room to evolve the system toward production-grade robustness and scalability. Future iterations of the system can focus on strengthening observability, improving resilience patterns, and refining the data and caching layers to better support large-scale, real-world deployments.

The system could be further enhanced with:

-Distributed tracing for end-to-end debugging:
Integrating tools such as OpenTelemetry and Jaeger would allow us to trace a single request as it flows across Reservation, Order, Payment, and Inventory services. This

would make it easier to diagnose performance bottlenecks, identify slow services, and understand the impact of new changes on the overall request path.

-More efficient retry and backoff mechanisms:
Refining the retry policies at both the HTTP and message-broker levels, using exponential backoff with jitter and strict idempotency guarantees, would reduce unnecessary load during partial failures and avoid "retry storms" when downstream components are temporarily slow or unavailable.

-Improved dead-letter queue handling:
Enhancing the RabbitMQ dead-letter strategy with better routing, alerting, and automated replay workflows would ensure that failed messages are not only captured but also inspected, categorized, and, when safe, reprocessed without manual intervention.

-Database-per-microservice pattern for stronger isolation:
Aligning with the "database per microservice" design pattern by giving each core service (Reservation, Order, Payment, Inventory, User Management) its own dedicated database would reduce coupling at the persistence layer. This would improve scalability, allow independent schema evolution, and minimize the blast radius of failures or heavy queries in one domain affecting others.

-Separating read-heavy and write-heavy workloads:
Splitting the data layer into logical read and write paths would enhance performance under flash-sale traffic. This could include using read replicas for queries, write-optimized stores for transactional operations, or even adopting a CQRS-style approach where command (write) and query (read) responsibilities are handled by different models and possibly different databases.

-Sharded inventory caching for extreme performance gains:
Partitioning inventory keys across multiple Redis instances or shards—based on product category, region, or sale event—would reduce contention on hot keys during peak moments. This would further decrease latency and allow the system to scale to larger catalog sizes and higher concurrency.

-Autoscaling and capacity planning in containerized environments:
Containerizing all services (e.g., using Docker and Kubernetes) and enabling horizontal pod autoscaling based on CPU, memory, or custom metrics (like queue depth or request rate) would allow the system to react dynamically to traffic spikes, scaling out during flash sales and scaling in during quieter periods.

Together, these improvements would move the system beyond a training-oriented implementation toward a mature, cloud-ready flash sale platform that can sustain heavy, geographically distributed traffic while preserving reliability, observability, and maintainability.