Performance Analysis and Refactoring of the Synapse Reference Server through Observability

Francisco A. A. Gomes[1], Francisco A. Silva[2], Michel Sales[1], Windson Viana[1], Fernando A. M. Trinta[1], Rossana M. de C. Andrade[1], José A. F. de Macêdo[1], Vinícius Lagrota[3], Rodrigo Pacheco[3], Paulo A. L. Rego[1]

[1]Universidade Federal do Ceará (UFC)

[2]Universidade Federal do Piauí (UFPI)

[3]Centro de Pesquisa e Desenvolvimento para a Segurança das Comunicações (CEPESC)

almada@crateus.ufc.br michelsb@ufc.br windson@virtual.ufc.br

{fernando.trinta,jose.macedo,paulo,rossana}@dc.ufc.br

faps@ufpi.edu.br {vinicius.1232,rodrigo.6874}@abin.gov.br

Abstract. *The Matrix protocol is a leading solution for decentralized, real-time communication, with Synapse as its primary reference server. Despite its wide adoption, systematic performance evaluations leveraging observability have been limited. This study instruments Synapse with distributed tracing, metrics collection, and network traffic analysis to characterize endpoint usage, identify bottlenecks, and guide architectural optimization. Our analysis revealed that synchronization, database contention, and lock management are the primary sources of latency, particularly around the* `/sync` *and message dispatch endpoints. Guided by observability insights, we applied targeted refactoring, including cache resizing, workload redistribution to specialized workers, bare-metal deployment, PostgreSQL tuning, and connection pooling. These interventions transformed Synapse into a highly scalable system, increasing throughput from 11 to 405 requests per second and reducing average response time from 4755 ms to 92 ms. The results demonstrate that observability not only diagnoses performance issues but can directly inform refactoring strategies to achieve substantial operational gains in complex, distributed communication platforms.*

1. Introduction

Synapse is the reference implementation of the Matrix protocol [Martins et al. 2025], an open standard designed for decentralized communication that enables real-time messaging, voice, and video across a federated network of servers. Developed in Python, it offers client–server and server–server APIs, supporting end-to-end encryption, and message synchronization, which makes performance and scalability central concerns. Beyond its technical design, Matrix has been adopted in both public and private sectors, including governmental organizations in Germany, France, Sweden, and Luxembourg, as well as within the free and open-source ecosystem through initiatives such as Mozilla, and KDE [Martins et al. 2026]. With more than 80 million reported users and active discussion within the IETF on interoperability standards, Matrix represents a significant case study in the deployment and evolution of decentralized communication infrastructures [Albrecht et al. 2024].

Synapse instances consist of multiple interdependent components that can be conceptualized as microservices, endowing the system with features of complex cloud-native applications such as high distribution, dynamic behavior, and unpredictability. This distributed nature increases the likelihood of failures and performance degradation, making system diagnosis particularly challenging, especially when many component instances operate concurrently [Dragoni et al. 2017]. Conventional monitoring solutions often fail to provide comprehensive visibility and cannot proactively detect deviations from expected behavior before they affect users.

Such challenges have driven the embrace of observability, an advanced form of monitoring intended to provide actionable insights into system behavior [Kosińska et al. 2023]. Observability refers to the ability to deduce the internal state of a complex system from its externally visible signals [Usman et al. 2022], consolidating metrics, logs, and traces into a coherent view of how requests traverse components, where time is spent, and which resources are under pressure. Unlike traditional monitoring, which often focuses on predefined indicators and coarse alarms, observability supports exploratory investigation, allowing operators to ask new questions as the system evolves and workloads shift. In practice, it enables administrators and developers to continuously compare the system's runtime behavior to its expected operation across the software lifecycle, from deployment and scaling decisions to incident response and postmortem analysis. As a result, observability promotes informed performance optimization, strengthens reliability by surfacing hidden dependencies and contention points, and supports proactive issue detection before degradations become visible to end users [Karumuri et al. 2021].

By combining observability techniques with load testing, we identified critical inefficiencies in Synapse's database management layer. The collected metrics and traces revealed that a substantial portion of request latency is not associated with endpoint-specific logic, but with recurring delays in acquiring database connections, scheduling queries, and coordinating concurrent access to shared state. In particular, Synapse relies on asynchronous data consistency mechanisms that, under concurrent workloads, frequently trigger locking at the persistence layer and amplify contention among worker threads and processes. This behavior creates cascading delays: lock acquisition stalls request handling, queued operations accumulate, and the system becomes increasingly sensitive to bursts of traffic and database growth. Consequently, throughput fails to scale with added concurrency, while response times increase sharply, indicating that the observed degradation is rooted in structural bottlenecks rather than sporadic anomalies.

To address these issues, the system requires architectural refactoring focused on reducing contention within the data layer. This may involve database tuning strategies, such as connection pool optimization as well as improvements in caching mechanisms within the application itself to minimize unnecessary database access. Such measures are essential to ensure that Synapse can evolve from a functionally correct system into a truly scalable and resilient platform. In this context, this study aims to address the following research questions:

RQ1 *How is observability leveraged to characterize the usage patterns of Synapse's endpoints?*

RQ2 *What is the impact of observability-driven insights on architectural refactoring aimed at improving Synapse's scalability?*

Overall, the study reinforces the role of observability not only as a diagnostic mechanism but as a strategic tool for continuous system evolution.

The remainder of this paper is organized as follows. Section 2 discusses related work on Matrix/Synapse and observability in distributed systems. Section 3 describes the methodology, including the instrumentation setup and the experimental procedure. Sections 4 and 5 present the performance analysis and bottleneck identification, followed by the optimization strategy and the impact assessment. Finally, Section 6 concludes the paper and outlines threats to validity and directions for future work.

2. Synapse Homeserver

Synapse is a server implementation of the Matrix protocol designed to support decentralized, secure, and low-latency communication among distributed users. It operates as the core component responsible for managing messages, room states, presence information, devices, and trust relationships across different servers. By avoiding reliance on a centralized infrastructure, Synapse enables independent organizations to run their own servers while still participating in a federated communication network [Li et al. 2023].

The architecture of Synapse is organized into well-defined layers, with a strong dependence on the persistence layer to ensure data durability and consistency. PostgreSQL is typically used as the primary database, storing large volumes of events, message histories, and room states. This design allows clients to retrieve information even after extended periods of disconnection, but it also makes the persistence layer a critical point, as it concentrates intensive read and write operations under high activity levels. One of the most performance-sensitive components of Synapse is the synchronization mechanism exposed through the /sync API. This endpoint is responsible for delivering incremental updates to clients, aggregating recent events, state changes, and relevant metadata. The process involves complex database queries, joins, and filtering operations, making system performance highly dependent on client usage patterns and the degree of concurrent access [Bilyatdinov et al. 2026].

To handle higher workloads, Synapse supports task distribution through specialized workers combined with load balancing strategies. This separation of responsibilities allows functions such as federation, synchronization, and event processing to be isolated across different processes. When used together with reverse proxies and connection pooling mechanisms, this approach helps reduce contention at the database layer and improves overall concurrency handling. Federation is another fundamental aspect of Synapse, enabling independent servers to exchange events and maintain shared rooms across different domains. While this capability significantly extends the reach of communication, it also introduces additional challenges, including variable latency, eventual consistency, and increased internal traffic. Asynchronous coordination between servers requires careful management to preserve global state coherence, particularly in geographically distributed environments [Li et al. 2023].

Finally, Synapse has been studied due to its operational complexity; however, the use of observability as a systematic approach for performance analysis remains limited. Instrumentation using metrics, logs, and traces enables a deeper understanding of system behavior, allowing the identification of recurring bottlenecks and the evaluation of different architectural configurations. These analyses provide a solid foundation for infrastructure tuning, parameter adjustment, and the continuous evolution of Synapse in

demanding production scenarios.

3. Related Work

Six papers were identified as having synergy with the objectives of this paper. Three papers focus on user distribution across rooms and servers [Jacob et al. 2019, Jacob et al. 2021, Li et al. 2023]. Jacob et al. [Jacob et al. 2019] evaluated the scalability of Matrix, a message-driven data synchronization middleware designed for federated platforms supporting decentralized near-real-time applications. The study offers a comprehensive understanding of performance and scalability characteristics. Jacob et al. [Jacob et al. 2021] explored the Matrix Event Graph (MEG) as a replicated data type for decentralized applications. The work validates MEG as a Convergent and Replicated Data Type (CRDT), ensuring strong eventual consistency in distributed systems. Li et al. [Li et al. 2023] analyzed the structure and security vulnerabilities of the Matrix network through in-depth exploratory research, focusing on homeservers, rooms, and users while exposing challenges related to decentralization, cybersecurity, and cryptographic protocols.

Three other papers emphasize security and privacy in Matrix [Müller et al. 2021, Riutort Grande 2021, Albrecht et al. 2023]. Muller et al. [Müller et al. 2021] examined the integration of informal communications into digital shop floor management systems, aiming to structure these exchanges for improved issue detection and resolution. Riutort et al. [Riutort Grande 2021] proposed a TOR-based privacy-preserving framework for IoT environments, enabling anonymous communication in devices with limited processing capacity. Albrecht et al. [Albrecht et al. 2023] revealed exploitable vulnerabilities in the Matrix protocol and the Element client, highlighting weaknesses in confidentiality and authentication guarantees when exposed to malicious servers. Regarding the use of Observability, some studies aim to contextualize the topic but do not provide specific information about its application in Synapse. For instance, Li et al. [Li et al. 2022] investigated microservices and observability, emphasizing the complexity of industrial microservice systems deployed in intricate cloud environments, where service instances are dynamically created and terminated. Their work highlights that microservice tracing and analysis represent a new big data challenge for software engineering, while also unveiling opportunities such as adaptive log sampling, data fusion for trace analysis, intelligent trace processing, and business intelligence derived from trace data. Similarly, Usman et al. [Usman et al. 2022] conducted a comprehensive study on distributed IT environments and microservices observability, aiming to identify challenges, requirements, best practices, and existing solutions for monitoring distributed systems. Moreover, the authors underscore the importance of organizational culture and individual mindsets in influencing the adoption of emerging observability tools and practices.

Table 1 compares the literature using five direct dimensions. *Matrix scope* states which layer each work targets (federation topology, room-level MEG replication, protocol/client security, or domain adoption). *Method* shows how evidence is obtained (large-scale measurement, formal/stochastic analysis, architectural design, integration, or practical exploitation). *Key signals/outputs* lists what each study delivers (e.g., skew/distribution patterns, SEC/CRDT and graph dynamics, threat surfaces and attack conditions, privacy trade-offs, or structured communication artifacts). *Main axis* labels the dominant concern (scalability, consistency, security, privacy, integration).

Unlike previous studies, which primarily focus on scalability concerns of the Ma-

trix protocol or explore security and privacy aspects, this work advances the state of the art by employing observability as a systematic method for performance evaluation in the Synapse reference server. Instead of limiting the analysis to protocol design or threat surfaces, we leverage observability signals to identify real usage patterns, reveal scalability constraints in database and concurrency management, and determine the exact conditions under which performance degradation occurs. This approach perspective not only exposes operational bottlenecks, but also provides concrete guidance for architectural refactoring and optimization that are not addressed in prior research.

Table 1. Related work comparison.

Ref.	Matrix scope	Method	Key signals / outputs	Main axis
[Jacob et al. 2019]	Federation-wide user/room/server distribution	Empirical measurement + large-scale analysis of federation structure	Load concentration patterns; room/server skew; scalability characterization	Scalability
[Jacob et al. 2021]	Matrix Event Graph (MEG) per room; replication and convergence	Formal analysis + stochastic modeling (Markov chains) of MEG dynamics	SEC/CRDT validation; graph width / forward-extremities behavior	Consistency
[Li et al. 2023]	Network ecosystem (homeservers, rooms, users) + security posture	Large-scale measurement (MatSpider) + exploratory security analysis	Distribution evidence; exposed threat surfaces and risks	Security
[Müller et al. 2021]	Matrix-based informal communication in industrial shop floor	Integration + extraction/structuring of communications for issue handling	Structured communication artifacts for detection/resolution workflows	Integration
[Riutort Grande 2021]	Privacy-preserving communication for constrained IoT (TOR-based)	Architecture proposal + feasibility discussion in constrained devices	Anonymity/privacy properties; overhead/constraints tradeoffs	Privacy
[Albrecht et al. 2023]	Matrix protocol + Element client under malicious server assumptions	Practical protocol/client security analysis and exploitation scenarios	Breaks in confidentiality/authentication under adversarial conditions	Security
This work	Synapse reference homeserver under realistic workloads	Observability-driven performance evaluation + load testing	Metrics/logs/traces to pinpoint DB/concurrency bottlenecks; degradation conditions	Performance

4. Methodology

In this study we have used the Synapse v1.122.0. To address the research questions, we organized the study into five main stages. In the **first stage**, we conducted a load test to identify performance bottlenecks and instances of low scalability in the Synapse server. The **second stage** involved applying observability techniques (e.g., distributed tracing) to characterize the usage patterns of Synapse's endpoints and uncover the root causes

of performance degradation, aiming to address *RQ1*. In the **third stage**, we analyzed the collected traces and identified that the main source of latency originated from inefficiencies in the database layer. The **fourth stage** consisted of an architectural refactoring process, in which we adjusted configurations and introduced new components to improve performance and scalability. Finally, in the **fifth stage**, we conducted a new load test to evaluate the impact of these changes, providing evidence to answer *RQ2* regarding the effectiveness of observability-driven insights for improving Synapse's scalability.

4.1. Materials and Methods

In this study, Synapse was deployed on a Kubernetes cluster to evaluate its behavior in a cloud-native, microservices-based environment, where dedicated resources were used to minimize interference from external processes. Figure 1 shows an overview of all tools and Synapse clients involved. Prometheus[1] was integrated to collect time-series metrics from the Synapse application, providing detailed insights into database and cache performance. Synapse natively exports metrics related to database operations, events, and cache usage with built-in support for Prometheus, simplifying monitoring setup.

Complementing this, Kubeshark[2] was employed to capture and analyze real-time network traffic across pods and services, including ingress, egress, and internal communications for protocols such as HTTP, gRPC, Redis, and Kafka, without requiring any code instrumentation. This enabled the identification of the most frequently used endpoints and an understanding of service interaction patterns.

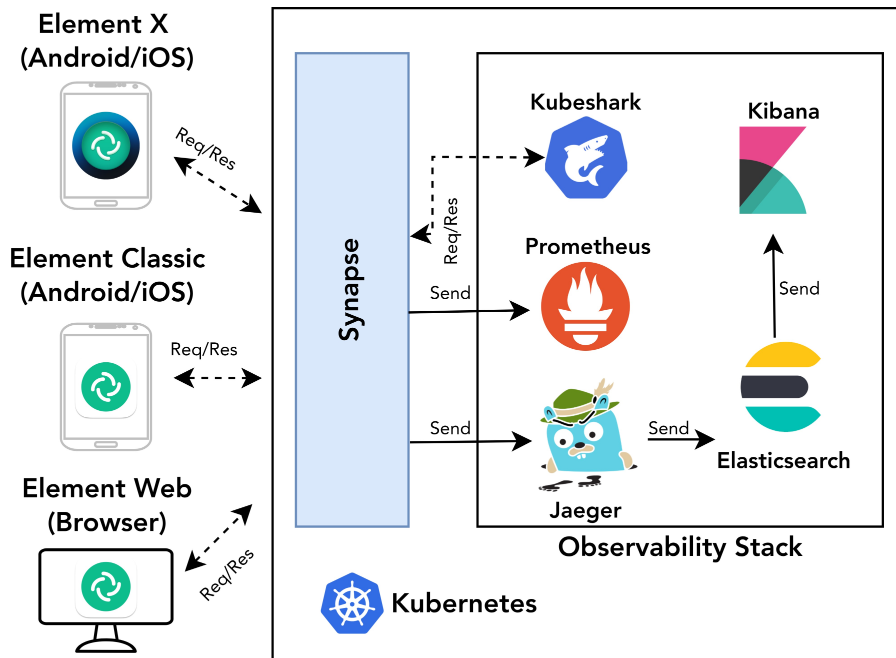

Figure 1. Observability Architecture for Synapse

In addition, Synapse includes instrumentation for distributed traces and offers native support for exporting them to Jaeger[3], enabling end-to-end tracking of requests across

[1] `https://prometheus.io/`
[2] `https://www.kubeshark.co/`
[3] `https://www.jaegertracing.io/`

multiple microservices and identification of the most time-consuming operations. All traces collected by Jaeger were stored in Elasticsearch[4], which provides efficient indexing and query capabilities, and visualized through Kibana[5] dashboards to facilitate interactive exploration and deeper comprehension of Synapse's internal behavior. The integrated use of Kubeshark, Jaeger, Prometheus, Elasticsearch, and Kibana enabled both metric and trace-level analysis of the system, supporting performance optimization and offering a robust framework for monitoring and understanding Synapse deployments in dynamic, distributed Kubernetes environments.

5. Workload Characterization and Baseline Analysis

Through Kubeshark, we monitored traffic within the Kubernetes cluster and create a dataset (made available in the Artifact Availability section) containing the requests, including their method, path, and timestamp. The traffic was generated by approximately 50 real users over the course of one week in an industry project using Synapse, whose name is omitted for confidentiality. The Table 2 presents the 20 most frequent requests.

Table 2. The 20 most frequent requests in the dataset.

Request	Count
GET /_matrix/client/v3/sync	441089
GET /_matrix/client/r0/sync	122327
GET /_matrix/media/r0/thumbnail/	48991
POST /_matrix/client/unstable/org.matrix.simpl...	25166
GET /_matrix/client/v1/media/thumbnail/	14585
GET /_matrix/client/versions	14085
PUT /_matrix/client/v3/rooms/typing	10117
POST /_matrix/client/r0/keys/query	7333
PUT /_matrix/client/r0/rooms/typing	5904
POST /_matrix/client/r0/rooms/read_markers	5596
POST /_matrix/client/v3/rooms/read_markers	5554
GET /_matrix/client/r0/devices/	5386
POST /_matrix/client/v3/keys/query	4377
POST /_matrix/client/v3/rooms/receipt/m.read/	4038
GET /_matrix/client/r0/devices	3496
PUT /_matrix/client/v3/rooms/send	3144
POST /_matrix/push/v1/notify	2685
PUT /_matrix/client/v3/user/account_data/im.ve...	2250
GET /_matrix/client/r0/pushers	2142
PUT /_matrix/client/r0/rooms/send	2048

From Table 2, we observe the relevance of the `/Sync` functionality to the operation of communication applications that rely on Synapse. This functionality represents one of the core components of the Matrix protocol, as it ensures that clients remain updated with the state changes occurring in the rooms they participate in. The `/Sync` mechanism enables clients to receive the latest messages sent by other users, join and leave events, presence updates, and additional information required to maintain a consistent and up-to-date timeline between server and client. Its operation resembles a polling strategy with a maximum timeout (30s), where the client periodically issues requests to the server

[4]`https://www.elastic.co/elasticsearch`
[5]`https://www.elastic.co/kibana`

in order to retrieve recent updates. Table 2 highlights three different *Sync* endpoints (R0, V3, and MSC3575), with the latter corresponding to the most frequent `POST` request (i.e., `/unstable/org.matrix.simplified_msc3575`). These versions represent distinct synchronization implementations, depending on the Matrix client adopted (e.g., Element relies on R0, Element Web on V3, and Element X on the simplified MSC3575).

6. Performance Analysis and Bottleneck Identification

Before conducting any synthetic stress tests, we examined system health using observability-driven metrics collected under normal runtime conditions. One of the most revealing indicators was the average time spent waiting for a database connection, collected by Prometheus, where the main Synapse process exhibited peaks of up to 27.9 ms. Since acquiring a database connection is among the very first steps of any operation, such delays inevitably cascade through the remaining execution stages, degrading performance across the board. Similarly, high database scheduling time quantiles exposed by Synapse and collected by Prometheus, representing the time required for database queries to be scheduled, peaked at 473 ms in the main process. These values provide strong evidence that responsiveness is being hindered primarily by contention at the database layer rather than by application-level logic.

To assess how this latent bottleneck manifests under user concurrency, we executed a load test using Locust[6], simulating repeated message-sending operations. The experimental machine is equipped with an 8-core Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz, 32 GB of RAM, and 100 GB of disk space. The baseline execution, performed without any tuning or architectural refinement, revealed a concerning fragility: with only 20 concurrent users, the system sustained a throughput of 11 requests per second, while the average response time escalated to 4755 milliseconds, a value fundamentally incompatible with real-time interaction requirements.

These observations indicate that the performance degradation is structural rather than incidental. Instead of applying incremental fixes without direction, we conducted a deeper inspection of the execution model, which showed that the main process frequently faced excessive waiting time and lock contention among worker threads distributed across Kubernetes pods. In highly threaded environments, competition for the Global Interpreter Lock (GIL) further constrained parallelism and introduced latency. Message dispatching also triggered a disproportionate number of interactions with Redis and the relational database, often issuing multiple queries for a single message. Although persistent database connections were intended to reduce overhead, under load they increased contention. As the database state grew, query execution times degraded proportionally, reinforcing the perception that Synapse aggressively consumes database resources and slows down as stored data accumulates.

7. Observability-Driven Architectural Refactoring

Having confirmed through observability metrics, execution traces, and empirical load testing that database contention was the dominant constraint on scalability, the next phase focused on a comprehensive architectural refactoring rather than superficial tuning. The traces were particularly valuable in pinpointing which operations contributed most to latency, providing granular visibility that complemented the high-level metrics. This archi-

[6] `https://locust.io/`

tectural approach aimed to systematically redistribute workload, optimize resource utilization, and reduce contention across the system.

Key interventions included resizing Synapse's internal cache structures to minimize redundant database lookups and reduce read amplification. Many responsibilities previously handled by the main process were offloaded to specialized workers, such as background workers, media handlers, application bridges, user directory processors, pushers, event persisters, and generic workers configured in parallel. By isolating these components into dedicated execution units, contention around shared resources was significantly reduced. Both the main process and the worker pool were deployed as bare-metal services managed via `systemd`, removing overhead from container orchestration layers. Inter-process communication was optimized using UNIX domain sockets for replication, replacing HTTP and Redis intermediations, and hostname resolution was eliminated from the critical path by directly using IP addresses to avoid DNS-induced stalls.

On the persistence layer, PostgreSQL was carefully tuned to sustain high levels of write concurrency, while PgBouncer was deployed as a lightweight connection-pooling proxy in front of the database. This setup enabled efficient reuse of established sessions, mitigating connection storms under bursty workloads and lowering transaction setup overhead, an effect that is particularly noticeable for short-lived interactions that would otherwise spend a disproportionate amount of time on connection negotiation. In addition to stabilizing database access patterns, the configuration helped smooth resource consumption at the database tier (e.g., backends, memory, and lock management), improving overall predictability during peak synchronization periods. Furthermore, synchronization traffic was load-balanced across the fleet of generic workers so that update propagation remained evenly distributed, avoiding hotspots where requests could concentrate on a single worker or thread of execution. As a result, the system reduced queue buildup during propagation waves and maintained steadier throughput, even when multiple clients issued concurrent updates and the persistence layer was under sustained pressure.

8. Impact Assessment

This section assesses the impact of the proposed refactoring by comparing Synapse's performance before and after the observability-driven interventions, focusing on scalability, throughput, and response time under load, which capture both system efficiency and user-perceived latency. After the Synapse optimization, we rerun the same load test, the transformation was substantial. With 20 concurrent users, throughput increased from 11 to 405 requests per second, while the average response time dropped from 4755 ms to just 92 ms. Execution traces collected after optimization confirmed that the operations that previously dominated latency no longer appeared as persistent outliers, validating that the architectural interventions effectively neutralized the critical bottlenecks. Additionally, the average time spent waiting for a database connection dropped to 1.06 ms, and the database scheduling time quantiles exposed by Synapse reached peaks of 10.6 ms. Thus, by also leveraging these metrics, it was possible to verify a significant improvement in database management indicators.

These results indicate that the performance bottlenecks were rooted in the system's architecture rather than being occasional anomalies. They also show how observability, through both metrics and traces, can effectively support architectural decisions. Grounded in concrete evidence, the changes were focused and meaningful, turning an execution pipeline that initially struggled under load into a distributed, scalable, and high-

throughput messaging core capable of meeting the demands of modern communication workloads.

8.1. Tracing Evaluation

Using the Element Web, the instrumentation of Synapse using Jaeger was carried out in a controlled experiment where two users, to simulate message exchange, exchanged approximately two hundred messages in a local Matrix instance. This setup ensured a realistic but reproducible workload, generating sufficient activity to expose the internal dynamics of the server. By leveraging spans already instrumented in the Synapse codebase, we collected a comprehensive set of operations executed during message handling, synchronization, and background processing. Two rounds of measurement were performed: one before and one after the refactoring process. This comparison allowed for a quantitative assessment of performance improvements and identification of which components benefited most from the architectural and tuning changes.

While the raw list of spans is extensive, grouping them by functionality allows us to identify the most critical areas of overhead and the types of work performed by the server. Operations are divided into categories such as REST services, caching and synchronization, database queries, and concurrency control. For each group, we report average execution times before and after refactoring, along with the factor by which the execution time decreased, and discuss their implications for performance and scalability. This layered view highlights where bottlenecks occurred prior to refactoring and how the optimization efforts contributed to more efficient processing.

8.1.1. REST Services: Entry Points for Client Requests

This group encompasses the main operations directly exposed to external clients through HTTP requests (see Table 3). Each servlet handles a different type of request, from synchronization of timelines to sending new events or updating profile information. Since these operations mark the boundary between the client and the server, their efficiency directly determines the responsiveness perceived by end users.

Table 3. REST service operations before and after refactoring.

Operation	Before (ms)	After (ms)	Times Faster
SyncRestServlet (with timeout)	10,582	10,422	1.02x
SyncRestServlet (without timeout)	56.0	39.1	1.43x
RoomSendEventRestServlet	282.7	165.3	1.71x
ReadMarkerRestServlet	216.3	47.2	4.58x
RoomTypingRestServlet	22.8	14.2	1.61x

The results confirm that the `SyncRestServlet` continues to dominate system activity. This servlet is requested by the Sync routes R0 and V3, as indicated in Table 2. With an average execution of 10 s (with timeout) and 56 ms (without timeout), this operation reflects the inherent cost of maintaining long-poll connections at the `/sync` endpoint. The span duration includes both waiting for events and preparing responses, which explains why it is significantly longer than other endpoints. After the refactoring, this servlet did not show any meaningful improvement in the long-poll (with timeout)

variant, as its duration is mostly determined by the expected waiting time for new events rather than by server processing. However, in the non-blocking variant (without timeout), execution time decreased by approximately 1.43 times, indicating that the refactoring effectively reduced overhead related to message preparation and event processing.

Other endpoints benefited more substantially. The `RoomSendEventRestServlet`, responsible for accepting and propagating new events, became 1.71 times faster, while the `ReadMarkerRestServlet`, responsible for updating the read position of a user in a room, improved 4.58 times, reflecting a substantial reduction in database and cache processing overhead. Similarly, the `RoomTypingRestServlet`, responsible for notifying that a user is typing in a room, became 1.61 times faster, which can be attributed to lighter synchronization mechanisms and reduced contention on shared state.

8.1.2. Caching and Synchronization Processes

In addition to the REST-level spans, several operations represent the internal mechanisms that support synchronization (see Table 4). These include the use of caches, waiting loops for new events, and functions that prepare room data for delivery to clients. Because the sync endpoint is invoked so frequently, these operations strongly influence system scalability.

Table 4. Caching and synchronization-related operations before and after refactoring.

Operation	Before (ms)	After (ms)	Times Faster
ResponseCache[sync].calculate (with timeout)	10,142	10,052	1.01x
wait_for_events (with timeout)	10,070	10,032	1.01x
ResponseCache[sync].calculate (without timeout)	49.55	36.2	1.37x
wait_for_events (without timeout)	48.2	33.2	1.45x

The `ResponseCache[sync].calculate` span averages almost the same cost as the sync servlet itself, with and without timeout. This indicates that computing cache entries for sync responses is nearly as expensive as performing the synchronization without caching. In practice, this suggests that cache misses are extremely costly, and that further optimizations in response reuse could reduce load considerably. The `wait_for_events` operation, with an average of 10 s (with timeout), represents the blocking time during long-polling. This is not pure CPU time but waiting on external triggers, and yet it still ties up system resources and contributes to the high latency. In the without timeout case, the improvements were 1.37× and 1.45×, respectively, indicating that the refactoring enhanced the caching and synchronization process.

8.1.3. Database Operations

Database calls form the backbone of Synapse, as most features require reading or updating state in persistent storage (see Table 5). The traced operations reveal a wide range of performance characteristics. Some queries, especially those related to account data, take more than one second on average, while many others execute in a few milliseconds.

Table 5. Representative database operations before and after refactoring.

Operation	Before (ms)	After (ms)	Times Faster
db.has_completed_background_updates	23.2	4.2	5.52x
db.get_account_data_for_room_and_type	17.1	7.8	2.19x
db.simple_select_one	7.3	5.4	1.35x
db.is_server_admin	17.0	8.2	2.07x
db.connection	5.3	2.1	2.52x
db.query	3.1	1.2	2.58x

The most time-consuming, `db.has_completed_background_updates`, decreased from 23.2 ms to 4.2 ms, becoming approximately 5.5 times faster. Similarly, `db.get_account_data_for_room_and_type` improved from 17.1 ms to 7.8 ms (2.2 times faster), and `db.is_server_admin` decreased from 17.0 ms to 8.2 ms (2.1 times faster). Smaller operations also benefited: `db.simple_select_one` went from 7.3 ms to 5.4 ms (1.35 times faster), `db.connection` decreased from 5.3 ms to 2.1 ms (2.5 times faster), and generic queries (`db.query`) improved from 3.1 ms to 1.2 ms (2.6 times faster). These results indicate that the refactoring substantially enhanced the responsiveness of database operations. Both high-cost and routine queries execute faster, contributing to lower latency across the system and improving the overall efficiency of frequently executed tasks.

8.1.4. Locking and Concurrency Management

Concurrency management is critical in distributed systems like Synapse. Locks ensure that only one process modifies a shared resource at a time, but lock contention can delay operations and reduce throughput. Traced spans related to locking provide insight into where contention arises and how the refactoring affected performance.

Table 6. Locking and concurrency management operations before and after refactoring.

Operation	Before (ms)	After (ms)	Times Faster
WaitingLock.lock	181.6	112.3	1.62x
db.drop_lock	7.2	5.1	1.41x
WaitingLock.waiting_for_lock	10.8	5.6	1.93x
db.try_acquire_read_write_lock	10.5	3.2	3.28x

The data shows that acquiring locks is the most time-consuming operation, with `WaitingLock.lock` decreasing from 181.6 ms to 112.3 ms (1.62 times faster) after refactoring. Other lock-related operations also benefited: `db.drop_lock` improved 1.41 times, `WaitingLock.waiting_for_lock` became 1.93 times faster, and `db.try_acquire_read_write_lock` showed the largest improvement at 3.28 times faster. Overall, the refactoring effectively reduced contention and improved concurrency management.

9. Final Considerations

This study showed that observability, combining metrics and distributed traces, is essential to identify structural bottlenecks in Synapse, particularly in synchronization, database access, and concurrency management. Traces were crucial in pinpointing latency-critical operations, guiding architectural refactoring that transformed the system into a scalable, high-throughput messaging core. Optimizations such as cache resizing, workload redistribution to specialized workers, bare-metal deployment, UNIX socket communication, PostgreSQL tuning, and PGBouncer integration substantially improved performance, with throughput increasing from 11 to 405 requests per second and average response time dropping from 4755 ms to 92 ms.

9.1. Research Questions

Moreover, to directly address the research questions: for **RQ1**, observability, combining metrics, traces, and network traffic analysis, enabled us to identify that the `/sync` endpoint dominates system activity and contributes disproportionately to latency. Tracing revealed which internal operations, such as cache calculations and event waiting, consume the most time, while metrics highlighted bottlenecks in database access and lock contention. This demonstrates that observability can effectively characterize endpoint usage and pinpoint critical performance hotspots.

For **RQ2**, leveraging insights from traces and metrics allowed us to identify structural bottlenecks and design targeted interventions. Architectural refactoring included resizing caches, offloading work to specialized workers, deploying main and worker processes as bare-metal services, optimizing inter-process communication via UNIX sockets, tuning PostgreSQL, and introducing PGBouncer for connection pooling. These measures, guided by observability data, significantly improved throughput and reduced response times, confirming that observability can effectively inform scalability-focused architectural decisions.

9.2. Threats to Validity and Future Work

Despite our analysis, some threats to validity remain. Load tests and traces were conducted in a controlled Kubernetes setup with limited users, which may not reflect large-scale deployments. However, the identified bottlenecks and optimizations target structural limitations expected to persist under higher loads. Performance gains are specific to the Synapse version and configuration used, and external factors like network latency, disk I/O, or background workloads could affect the observed results. Future work includes exploring adaptive caching, finer-grained concurrency control, predictive observability for proactive performance management, and scalable worker orchestration to further enhance responsiveness and throughput in large Matrix deployments.

Artifact Availability

The dataset with the requests is available at: https://doi.org/10.5281/zenodo.19454911

References

Albrecht, M. R., Celi, S., Dowling, B., and Jones, D. (2023). Practically-exploitable cryptographic vulnerabilities in matrix. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 164–181. IEEE.

Albrecht, M. R., Dowling, B., and Jones, D. (2024). Device-oriented group messaging: a formal cryptographic analysis of matrix'core. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 2666–1685. IEEE.

Bilyatdinov, K., Kiselev, S., and Petukhova, A. (2026). Information security model for messengers based on matrix protocol. *International Journal of Open Information Technologies*, 14(1):102–108.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pages 195–216.

Jacob, F., Beer, C., Henze, N., and Hartenstein, H. (2021). Analysis of the matrix event graph replicated data type. *IEEE access*, 9:28317–28333.

Jacob, F., Grashöfer, J., and Hartenstein, H. (2019). A glimpse of the matrix (extended version): Scalability issues of a new message-oriented data synchronization middleware. *arXiv preprint arXiv:1910.06295*.

Karumuri, S., Solleza, F., Zdonik, S., and Tatbul, N. (2021). Towards observability data management at scale. *ACM SIGMOD Record*, 49(4):18–23.

Kosińska, J., Baliś, B., Konieczny, M., Malawski, M., and Zielinśki, S. (2023). Towards the observability of cloud-native applications: The overview of the state-of-the-art. *IEEE Access*.

Li, B., Peng, X., Xiang, Q., Wang, H., Xie, T., Sun, J., and Liu, X. (2022). Enjoy your observability: an industrial survey of microservice tracing and analysis. *Empirical Software Engineering*, 27:1–28.

Li, H., Wu, Y., Huang, R., Mi, X., Hu, C., and Guo, S. (2023). Demystifying decentralized matrix communication network: Ecosystem and security. In *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 260–267. IEEE.

Martins, J. A., Rego, P. A., de Macêdo, J. A., Silva, F. A., and Lagrota, V. (2026). Matrix protocol: a comprehensive systematic mapping study. *Journal of Cloud Computing*, 15(1):20.

Martins, J. A. P., Rego, P. A. L., Macêdo, J. A. F. d., Andrade, R. M. C., Bonfim, M. S., Ivo, R. F., Costa, V. L. R. d., Pacheco, R. P., and Silva, F. A. P. d. (2025). Protocolo matrix: Conceitos, arquitetura, aplicações e desafios. In *Jornada de Atualização em Informática 2025*. SBC, Porto Alegre, RS, Brasil.

Müller, M., Lee, J.-U., Frick, N., Stangier, L., Gurevych, I., and Metternich, J. (2021). Extracting problem related entities from production chats to enhance the data base for assistance functions on the shop floor. *Procedia CIRP*, 103:231–236.

Riutort Grande, P. (2021). How p2p framework can help mitigate trust and security risks of iot applications. *Procedia CIRP*.

Usman, M., Ferlin, S., Brunstrom, A., and Taheri, J. (2022). A survey on observability of distributed edge "&" container-based microservices. *IEEE Access*, 10:86904–86919.