TÉCNICO LISBOA

Handling high-throughput on a distributed system

Performance analysis of a user-oriented notification system at CERN

José Francisco Lopes da Silva Malanho Semedo

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Luís Manuel Antunes Veiga
Doctor Andreas Wagner

Examination Committee

Chairperson: Prof. João António Madeiras Pereira
Supervisor: Prof. Luís Manuel Antunes Veiga
Member of the Committee: Prof. João Nuno De Oliveira e Silva

May 2025

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First and foremost, I wish to express my deepest and most sincere gratitude to my supervisor, Professor Luís Antunes Veiga, for accepting to supervise and guide me through this project. His understanding during challenging periods, his welcoming approach and his consistent encouragement were instrumental to my progress. I genuinely believe that without a supervisor of such character to help bridge the gap between my academic anxiety and desire to complete this stage of my life, this work would not have been possible.

I am particularly indebted to my former supervisor at CERN, Andreas Wagner, for incentivizing and allowing me to pursue this final step in my degree. His willingness to make time, offer assistance, and remove obstacles along the way has been invaluable throughout this journey.

A heartfelt thank you to my former colleagues at CERN who welcomed me and created a sense of home when I was far from my own. Their support and camaraderie made a significant difference in my professional and personal experience.

I would not have found the courage to embark on this final academic step without the unwavering support of my dear partner, Bárbara Bessa. For your patience, encouragement, and belief in me, you have my sincerest appreciation, now and forever.

On a personal note, I am deeply grateful to my family, especially my parents, who have consistently supported and encouraged me in every path and decision I have taken. They have given me their shoulders to stand on so that I could reach further and higher than I ever believed I could myself.

This section would not be complete without acknowledging some of my closest friends who have supported me throughout this journey: João Marçal, Daniel Fermoselle, Tiago Rodrigues, Jaime Luz, Catarina Gomes, Alexandre Coelho, and Marta Fiolhais. Thank you for caring, for pushing me forward, and for celebrating my successes alongside me.

Abstract

As institutions scale in size and operational complexity, the need for responsive, targeted, and configurable communication systems becomes increasingly critical. The CERN Notifications System was designed to fulfill this role across CERN's diverse and high-demand environment, enabling multichannel, user-customizable notifications. However, performance limitations, particularly in scenarios involving large-scale message dissemination, threaten the system's responsiveness and scalability. This dissertation addresses these limitations through a focused performance analysis of the system's routing component, the segment responsible for message expansion, targeting logic, and delivery preparation.

To support this work, a detailed tracing-based performance analysis was conducted. Using OpenTelemetry for instrumentation and Jaeger as a backend, the system was profiled under controlled workloads simulating real-world notification patterns. This empirical evaluation provided insight into the system's runtime behavior, revealing areas of inefficiency and informing targeted optimization strategies.

Informed by the trace data, a set of prototype code-level optimization proposals was put forth. These include the introduction of caching mechanisms, parallel execution via thread pools, and the adoption of set data structures to replace list-based operations. Additionally, an outdated external API integration was modernized and parallelized to further reduce latency during group resolution.

The combined improvements were discussed for their expected impact on system performance and latency. This work strengthens the CERN Notifications System's ability to meet future demand and offers practical guidance on trace-driven optimization and instrumentation strategies in distributed, event-driven architectures.

Keywords

Distributed Systems; Pub/Sub; System Profiling; Tracing; Performance Optimization; Parallelism; Caching

Contents

List of Figures

List of Tables

Listings

1

Introduction

Contents

1.1 Context and Motivation

The CERN Notifications System was initially developed as part of the MALT project [1], which aimed to reduce CERN's dependency on commercial software and promote internally developed, flexible, and maintainable services. Within this context, the need arose for a unified, extensible platform to manage the dissemination of information across CERN's vast and diverse organizational landscape.

What began as a component initiative soon matured into a standalone service, CERN Notifications, dedicated to enabling structured, targeted communication across the institution. The service provides a centralized and programmable notification infrastructure that allows internal services and teams to deliver messages to individuals or groups through multiple channels, such as email, SMS, push, and messaging platforms.

CERN's large, dynamic, and heterogeneous community, comprising researchers, technical staff, support services, and external collaborators, generates a continuous and high-volume flow of information. Managing this information effectively is essential for productivity, coordination, and operational responsiveness. Traditional communication mechanisms, such as global emails or ad-hoc messaging, often suffer from a lack of targeting refinement, configurability, and overload control, which can result in messages being ignored, relevant information becoming buried, taking a toll on user focus.

The CERN Notifications System addresses this gap by offering a user-configurable interface where individuals can choose how, when, and through which devices they receive messages. This design not only improves user agency but also reduces information fatigue and facilitates organizational coordination.

The service exposes a documented and public API and empowers other internal applications and workflows to integrate the notification capabilities without the need to replicate core logic, streamlining development and promoting consistency across the ecosystem.

This has led to widespread adoption across CERN, with several teams relying on it for operational and user-facing communication. It has also effectively taken over duties from older systems which have been decommissioned, such as the CERN Alerter, positioning itself as the *de facto* standard for event-based messaging at the institution.

As its usage and importance continue to grow, so do the performance and scalability demands placed on it, particularly under scenarios that involve delivering time-sensitive messages to large user populations. This dissertation is motivated by the need to enhance the responsiveness and performance of the CERN Notifications system as a whole, particularly in scenarios requiring fast, large-scale message dissemination. Through analysis, a key contributor to system latency was identified, the routing component, making it the primary focus for performance analysis and design of optimizations in this work.

1.2 Problem Statement

While the CERN Notifications System has matured into a widely adopted and operationally critical service, the increasing volume, complexity, and urgency of communication use cases have revealed performance limitations that must be addressed to maintain responsiveness and reliability.

The architecture of the system is event-driven and modular, but one of its key components, the router, has become a bottleneck in scenarios involving large-scale message expansion and delivery. This layer is responsible for determining which users should receive a given notification. As usage patterns evolve to include high fan-out notifications and increasingly complex targeting logic, the current implementation begins to show signs of strain.

These issues do not manifest uniformly but become particularly problematic in time-sensitive events, such as safety alerts, system incidents, or emergency broadcasts, where delivery delays can undermine the purpose of the communication.

Thus, the core problem addressed in this dissertation is the routing component's inability, in its current form, to meet the system's evolving performance requirements, especially under critical workloads. Assessing the sources of bottlenecks in this layer and designing optimizations to address them is necessary to ensure that CERN Notifications remains scalable, responsive, and reliable as its operational footprint continues to expand.

1.3 Proposed Solution

This work proposes a structured, data-driven approach to designing performance optimization avenues for the CERN Notifications System, with a focus on optimizing the routing component, the most latency-sensitive part of the pipeline in large-scale notification scenarios.

Central to this effort is a comprehensive performance analysis, which forms the foundation of the proposed improvements. Leveraging distributed tracing, in particular through OpenTelemetry instrumentation and Jaeger-based visualization, detailed execution traces were collected across a range of test cases simulating real-world workloads. This process enabled the identification of key bottlenecks, such as expensive group resolution calls, redundant per-user operations, and serial execution paths that could be parallelized.

The proposed solution designs are not a redesign of the system architecture, but rather targeted optimizations of code paths based on empirical evidence. The core interventions include: instrumented trace collection and analysis, caching mechanism implementation, thread-based parallelization, switch to set-based operations, and reworking external service API endpoint usage.

These strategies were selected because they directly address the dominant contributors to latency

observed in the trace data. The optimizations are designed to preserve system correctness and compatibility while significantly improving the system's ability to handle time-sensitive, large fan-out notifications.

1.4 Contributions

This dissertation makes several contributions across analysis, optimization, and methodology, aimed at improving the performance and scalability of the CERN Notifications System:

- Tracing Technology Evaluation - This work includes an evaluation of tracing technologies suited for integration into the CERN Notifications System. Several distributed tracing frameworks and backends were taken into consideration, such as Jaeger, Zipkin, SigNoz, and others, alongside the OpenTelemetry standard. The decision to adopt OpenTelemetry and Jaeger was grounded in technical fit, openness, and alignment with CERN's broader ecosystem. This selection process serves as a reusable reference for similar instrumentation efforts.
- Controlled Tests for Execution Profiling - A series of structured test cases was developed to simulate different routing execution scenarios and workloads. These controlled tests enabled systematic profiling, supporting the identification of latency hotspots, such as those triggered by wide-reaching information dissemination.
- Code-Level Performance Optimization - Prototype solution proposals were designed to address several performance bottlenecks identified through trace analysis. Optimization techniques included caching (e.g., memoized group resolution), parallelism using threaded pools, and replacing inefficient data structures (e.g., sets over lists). These interventions directly targeted critical performance paths in the routing logic.
- Impact Analysis - Theoretical and practical performance implications were analyzed, particularly in the context of I/O-bound and multithreaded workloads. The discussion is framed around the demands of latency-sensitive systems, such as large-scale notification delivery under tight time constraints.
- Guidance for Future Work - This work exemplifies a structured approach to performance engineering in message-oriented systems. It also outlines considerations and directions for continuing performance-focused development.

1.5 Document Structure

The remainder of this document is organized as follows:

- Chapter 2 - Related Work surveys prior research relevant to this dissertation, focusing on notification delivery architectures and performance analysis techniques such as tracing and profiling.
- Chapter 3 - Current Architecture - CERN Notifications describes the CERN Notifications System with a focus on its component layout, data flow, and architectural considerations relevant to performance under high load.
- Chapter 4 - Performance Assessment and Analysis presents the experimental methodology and the assessment of tracing-based profiling under different workload scenarios. It identifies key latency contributors and behavioral patterns, and presents prototype solution proposals for each of the key issues identified.
- Chapter 5 - Conclusion and Future Work summarizes the key findings and contributions, and outlines opportunities for further optimization and research.

2

Related Work

Contents

Understanding the foundations and context of this work requires examining both the evolution of systems for information distribution and the methods available for analyzing and improving their performance. This chapter reviews previous efforts in the design of notification and publish-subscribe architectures, highlighting their strengths and limitations in supporting real-time, scalable delivery. It also surveys relevant research on how profiling and tracing tools can be used to assess system behavior and guide optimization. Together, these areas inform the technical choices and strategies applied in this project.

2.1 Web Syndication and Information Distribution

Web syndication refers to the practice of making digital content available for reuse and redistribution across different platforms or services. It enables the decentralized dissemination of information from a central source to multiple consumers, allowing users or applications to receive updates without manual intervention. This model is foundational for scalable content delivery on the internet and is commonly used in contexts such as news delivery, blogs, software updates, and notifications. Syndication protocols provide standardized formats for describing and exchanging content, forming the conceptual basis for technologies like RSS.

RSS

RSS (Really Simple Syndication) [2] emerged as one of the earliest methods for automated content distribution across the web. Designed primarily for syndicating blog updates and news headlines, RSS offered a lightweight and decentralized mechanism for publishing and consuming information. In the context of institutions like CERN, RSS was initially adopted due to its simplicity, open standards, and wide client support across operating systems and browsers. It required minimal infrastructure to implement, making it an attractive option for early-stage dissemination of institutional updates, alerts, and announcements.

CERN's first dedicated notification system, CERN Alerter, was built upon this model by leveraging structured polling mechanisms to notify users of updates. Although effective in its time, this approach soon revealed substantial limitations, particularly as the scale and complexity of communication needs at CERN grew.

Fundamentally, RSS operates on a pull-based architecture. Clients must periodically poll the server to check for updates, introducing latency and inefficiency. This polling interval is fixed on the client side and cannot adapt dynamically based on urgency or priority of information. As a result, time-sensitive notifications may be delayed until the client performs its next fetch cycle. This makes RSS poorly suited for real-time communication or alerting systems, where immediate delivery is critical.

Moreover, the polling model leads to wasteful resource usage; even in the absence of new updates, clients continue to query servers at regular intervals, consuming unnecessary bandwidth and compute resources on both the client and server side. This becomes increasingly problematic at scale, where thousands of clients may be checking the same feed repeatedly, creating artificial load on backend services.

RSS is also stateless and broadcast-oriented, meaning it does not provide built-in support for personalization, access control, or selective targeting of recipients. It lacks any mechanism to differentiate delivery based on user preferences, device types, or notification priorities. Every subscriber receives the same payload, regardless of relevance or context. In a heterogeneous institution like CERN, composed of various teams, hierarchies, and user roles, such a one-size-fits-all model quickly becomes inadequate.

Another limitation lies in platform integration. While initially implemented with a Windows-centric design, CERN Alerter [3] eventually encountered challenges adapting to a modern, cross-platform computing environment. As Linux, macOS, and mobile platforms became more prevalent among CERN personnel, maintaining compatibility and consistent behavior across systems became increasingly complex.

Together, these shortcomings underscore the need for a more reactive, scalable, and user-configurable notification framework [4]. The shift away from RSS-based polling towards push-based, event-driven mechanisms reflects a broader industry trend that prioritizes responsiveness, precision, and flexibility, attributes that have become essential in modern messaging and alerting systems.

Publish-Subscribe Systems and Event-Driven Architectures

As communication requirements within large-scale, dynamic environments like CERN outgrew the limitations of polling-based approaches, publish-subscribe (pub/sub) systems emerged as a more scalable and flexible alternative for distributing information [5]. Unlike RSS, which relies on periodic client-side polling, pub/sub architectures support push-based messaging, where publishers emit messages that are immediately propagated to interested subscribers. This decoupling of producers and consumers of data is a defining characteristic that makes pub/sub systems inherently more scalable and responsive.

In the topic-based pub/sub model, which is the most widely adopted variant, messages are categorized under named topics. Subscribers express interest in specific topics, and the system ensures that only messages related to those topics are delivered to them. This model is exemplified by systems such as MQTT, Apache Kafka, and Google Cloud Pub/Sub [6]. These systems are optimized for high-throughput event streaming and offer features such as message retention, delivery guarantees, and consumer group coordination. While highly performant, topic-based systems are limited in their expressiveness, they require the publisher and subscriber to agree *a priori* on the topic structure and cannot perform dynamic, context-aware content filtering.

To address this, content-based pub/sub systems were proposed [7]. In this model, subscribers specify conditions over message content itself, rather than subscribing to predefined topics. The system evaluates these conditions at runtime and delivers only messages that satisfy them. Content-based pub/sub introduces significantly more computational overhead, particularly at the broker, which must evaluate each message against potentially complex subscriber predicates. As a result, while more flexible, content-based systems often suffer from reduced throughput and increased latency, especially in high-volume environments.

More advanced implementations, such as distributed event notification services, take these models further by distributing the pub/sub infrastructure across nodes to enhance fault tolerance, scalability, and locality. Google Pub/Sub, for example, offers a globally distributed messaging system that supports event-driven architectures with at-least-once delivery semantics and high availability guarantees.

In the context of CERN, pub/sub architectures offer clear advantages for systems like CERN Notifications. They allow decoupled services to interact asynchronously, enabling modular design and improved resilience. The system can support diverse publishers, ranging from internal applications to monitoring agents, without requiring explicit knowledge of downstream subscribers. Similarly, multiple consumers (e.g., email delivery agents, SMS services, push notification modules) can independently subscribe to notification events without impacting core logic.

However, it is also important to recognize the limitations of off-the-shelf pub/sub models when applied to more complex routing and user-preference scenarios. For example, CERN Notifications incorporates rich routing logic that takes into account user preferences, mutes, group membership intersections, and delivery context (e.g., device type priority). Such decisions often require stateful access to user metadata and context-aware processing that is difficult to express purely through pub/sub semantics.

Moreover, pub/sub systems typically treat all messages equally and do not inherently provide support for priority-based delivery, rate-limiting, auditing, or replay semantics that may be required in high-assurance notification systems. Queue-based or workflow-based processing layers are often required to fill this gap.

Consequently, while pub/sub architectures form a conceptual backbone for the notification delivery flow at CERN, they are complemented by custom routing components that interpret and act on message content, user state, and delivery policies in ways that exceed what is supported by traditional pub/sub infrastructure. This hybrid design reflects a pragmatic engineering approach: leveraging proven paradigms like pub/sub for transport and decoupling, while retaining flexibility through domain-specific routing logic.

WebSub: A Modern Push-Based Alternative

WebSub, formerly known as PubSubHubbub, is a standardized protocol developed by the W3C to enable real-time content delivery on the web using a push-based model [8]. It was introduced as a more modern alternative to RSS and Atom feeds, addressing the primary shortcomings of polling, namely latency, server load, and inefficiency. Instead of relying on clients to repeatedly check for new content, WebSub introduces a publish-subscribe mechanism using webhooks to notify subscribers as soon as new data is available.

The protocol operates with three core roles: the publisher, who owns the content (e.g., a website or service), the subscriber, who wants to be notified of updates, and the hub, which acts as a mediator between the two. When content changes, the publisher notifies the hub, which then send HTTP POST requests to all registered subscribers, delivering content directly to their endpoints. This architecture ensures timely delivery and significantly reduces redundant polling traffic, making WebSub an efficient and lightweight solution for real-time content syndication.

Despite its advantages, WebSub is not a viable candidate for addressing the requirements of a complex, institution-wide notification infrastructure like CERN Notifications. The system at CERN is designed not merely to push content, but to route messages intelligently, based on a rich set of user-defined rules, delivery contexts, group intersections, and dynamic filtering. WebSub provides no built-in support for such intermediate decision-making logic. Once the publisher emits an update, it is blindly delivered to all subscribers by the hub, with no opportunity for fine-grained control over who should receive the notification, how, or under what circumstances.

Another limitation of WebSub is its heavy reliance on HTTP endpoints as delivery channels. CERN Notifications must support a diverse range of delivery mechanisms, including SMS, email, push notifications, and message services (e.g., Mattermost), each with its protocol, failure behavior, and delivery guarantees. WebSub's webhook-only design is inherently for supporting non-HTTP clients or low-connectivity delivery contexts. For instance, delivering critical alerts via SMS to a subset of users requires access to specialized gateways, prioritization, and user-preference logic that falls far outside WebSub's design scope.

Additionally, security and access control are critical in enterprise-grade notification systems. While WebSub supports some verification mechanisms, it lacks the robust identity, permission, and auditing models required in environments like CERN, where group membership and channel ownership must be enforced rigorously. Integrating WebSub into such an ecosystem would necessitate extensive additional infrastructure to replicate these access checks.

From an engineering and deployment standpoint, WebSub also introduces centralization through its hub dependency, which can become a bottleneck or single point of failure. For systems aiming at high reliability, auditability, and fault isolation, this introduces operational risks that must be carefully mitigated.

CERN Notifications favors a pipeline-based architecture built around decoupled services and internal queues, which offer more flexible error handling and buffering capabilities than the direct webhook push model used in WebSub.

While WebSub represents a notable evolution in real-time web communication and addresses many of the inefficiencies of RSS, its simplified architecture and HTTP-centric delivery model make it poorly suited for high-complexity, multi-protocol, user-configurable notification systems like that in use at CERN.

The progression from early polling-based systems like RSS to modern publish-subscribe and push-based architectures reflects an industry-wide push toward real-time, scalable, and user-responsive communication models. RSS provided a simple entry point but lacked efficiency and adaptability for dynamic environments. Publish-subscribe models improved scalability and decoupling but often did not meet advanced delivery and routing needs. Similarly, WebSub introduced a real-time delivery via webhooks but fell short in supporting protocol diversity and dynamic user-side control. The CERN Notifications System draws inspiration from these foundational models, adopting the decoupling and scalability principles of pub/sub and the responsiveness of push-based architectures, while layering additional routing logic, user configurability, and protocol-specific handling to meet the institution's unique operational and technical requirements. In doing so, it bridges the gap between general-purpose messaging infrastructure and domain-specific platform tailored to CERN's communication demands.

2.2 System Profiling and Performance Optimization

Publish-subscribe systems must balance functionality with performance. While topic-based architectures [5] avoid the computational overhead of content-based filtering, advanced features like priority handling and multiprotocol delivery introduce additional broker-side logic. This work focuses on optimizing these routing mechanisms to reduce latency while maintaining system flexibility.

Effective optimization requires a structured approach:

- Performance characterization through instrumentation to identify critical paths
- Bottleneck analysis to distinguish essential operations from incidental overhead
- Targeted intervention using appropriate optimization techniques

Tracing is a widely adopted mechanism in modern systems for performance analysis. It enables developers to collect fine-grained temporal data about system behavior across services. The feasibility and benefits of distributed tracing in production environments have been recognized for some time, with foundational systems like Google's Dapper [9] laying the groundwork for modern tracing architectures.

Building upon this model, the OpenTelemetry project has emerged as the *de facto* industry standard for observability instrumentation. It is an open-source project under the Cloud Native Computing

Foundation (CNCF) and is actively maintained and adopted by a broad range of organizations, including Google, Microsoft, Amazon, and many others [10]. OpenTelemetry [11] provides vendor-agnostic APIs and SDKs for capturing metrics, logs, and traces, enabling comprehensive insight into system performance with minimal vendor lock-in.

To make use of tracing data, observability platforms provide storage, indexing, visualization, and query capabilities for distributed traces. These platforms allow developers to analyze the flow of execution across services, detect performance bottlenecks, and diagnose issues.

Commercial and managed observability solutions such as Splunk, Datadog, and Grafana Tempo offer integrated platforms combining tracing, logging, metrics, alerting, and automated anomaly detection. However, these solutions are not aligned with the open-source, interoperable, and vendor-neutral philosophy followed by this project and many initiatives at CERN. Consequently, commercial tools are not considered viable candidates for integration.

Instead, attention is given to open-source distributed tracing systems, which allow full control over deployment and integration. Numerous open-source options have been developed in recent years, varying in architecture, features, and maturity. A recent comparative study [12] reviews over 30 such tools and highlights the diversity in tracing capabilities and implementations.

Jaeger

Originally developed at Uber, Jaeger is a distributed tracing platform now maintained under the Cloud Native Computing Foundation (CNCF) [13]. It was created to support high-scale, production-grade tracing for microservices and event-driven systems. Jaeger offers a full set of features for trace ingestion, storage, visualization, and querying, along with built-in support for service dependency graphs and latency breakdowns. Jaeger's architecture is modular and flexible; it can operate with various backends (e.g., Elasticsearch, Kafka) and can scale horizontally, making it well-suited for large-scale environments.

Jaeger's native support for OpenTelemetry instrumentation is a critical asset. As OpenTelemetry becomes the standard for observability APIs and SDKs, Jaeger is designed to seamlessly ingest and visualize these traces. This compatibility reduces integration overhead and promotes future-proofing.

Its maturity, active community, and alignment with open standards made Jaeger a leading choice for integration into CERN's infrastructure, especially considering the existing familiarity and compatibility with other CNCF tools already deployed in the ecosystem.

Zipkin

Zipkin is one of the earliest distributed tracing systems, inspired by Google's Dapper and originally developed by Twitter [14]. It implements the core concepts of tracing, including span collection, propagation,

and visualization, using a simple architecture. While Zipkin is effective for tracing latency across microservices, its development has slowed relative to newer tools. Its support for OpenTelemetry is partial and not as comprehensive as that of Jaeger or SigNoz.

Zipkin excels in simplicity and low overhead, and can be easily deployed with minimal configuration. However, it lacks advanced features such as built-in support for high-cardinality tagging, dynamic sampling strategies, and flexible backend integrations. These limitations make it less suitable for high-scale or feature-rich observability pipelines, especially in heterogeneous infrastructures like CERN's.

Zipkin may still be useful for lightweight systems or development environments where overhead is a primary concern. However, for production-grade usage and advanced trace analytics, more modern systems offer richer functionality.

SigNoz

SigNoz is a relatively new open-source observability platform [15] built natively around OpenTelemetry. It positions itself as a full-stack solution for logs, metrics, and traces, aiming to be a drop-in alternative to proprietary platforms like Datadog or New Relic. SigNoz offers an integrated UI, supports structured logs and time-series metrics, and features a modern query language for trace analysis.

One of SigNoz's main advantages is its developer-oriented dashboard, which provides out-of-the-box insights into system performance. It simplifies the correlation of traces, metrics, and logs in one place, which can be helpful for unified observability workflows.

However, the platform is still maturing. Its community, while growing, is smaller than that of Jaeger, and certain features, such as alerting granularity, plugin support, may not yet be as robust. Given CERN's production reliability needs and the importance of long-term maintainability, SigNoz was considered promising but not yet stable enough for critical integration.

Among these solutions, Jaeger was evaluated as a strong candidate for integration within CERN's monitoring infrastructure [16]. Its scalability, open-source nature, and native support for OpenTelemetry make it well-suited for large-scale, distributed environments such as those at CERN. The choice of OpenTelemetry + Jaeger aligns with the engineering principles and pragmatic practices followed by this project and many others at CERN, namely, leveraging vendor-neutral, open technologies that are already present in the ecosystem. This not only reduces integration overhead but also facilitates long-term maintainability and adoption, as it builds upon technologies that are familiar and supported within the organization's infrastructure landscape.

Moreover, this evaluation and adoption process contributes to advancing the observability capabilities of the system, an increasingly critical attribute for operating reliable, complex infrastructures. Observability is more than just monitoring, it is about designing systems that enable engineers to understand

internal states through external outputs [17] [18]. By integrating tracing at the core of the routing logic's optimization workflow, this work reinforces observability as a first-class concern, supporting maintainability, ease of debugging, and future scalability.

The tracing-analysis methodology is further validated by recent studies that demonstrate how trace data can be used to gain actionable insights into system performance. Shahedi et al. (2024) [19] explore statistical models for profiling and regression detection using trace-based instrumentation, showing that performance-critical sections can be isolated without needing to analyze the entire trace corpus. Ezzati-Jivan et al. (2021) [20] introduce DepGraph, which uses software traces and dependency graphs to identify waiting dependencies and thread-level bottlenecks in multicore systems. It exemplifies how trace-based control flow analysis can expose concurrency bottlenecks and local performance issues within larger systems.

In microservices settings, Ibidunmoye et al. (2022) [21] applied NLP to distributed trace data to detect anomalies and performance outliers, reinforcing the analytical value of this telemetry stream. Techniques such as critical path tracing, as described by Alizadeh et al. (2022) [22], aggregate repeated trace patterns to identify the most impactful execution paths—those contributing most to latency across multiple transactions. Similarly, in data stream processing contexts, Ostermann et al. (2019) [23] demonstrate how adaptive tuning of execution behavior based on trace insights leads to meaningful performance gains.

Basing performance improvement efforts on tracing or profiling has become a recurring approach across both research and practice. By analyzing these traces, it becomes possible to identify hot paths—sections of code that consistently account for a disproportionate share of total execution time. Distributed tracing makes it possible to correlate latency spikes with specific inputs, decisions, or interactions across the stack, including synchronous blocking calls, external queues, databases, or APIs. This enables precise bottleneck identification and provides a principled foundation for optimization decisions.

This work follows that model, using trace data not only to observe behavior but also to guide and justify targeted optimization efforts.

Profiling and tracing have become indispensable for understanding performance in complex distributed systems. Modern approaches have moved beyond basic metric collection to embrace distributed tracing, which provides visibility into inter-service interactions, latency sources, and resource bottlenecks. Tools like OpenTelemetry and Jaeger offer vendor-neutral, extensible observability pipelines well-aligned with open infrastructure principles such as those followed at CERN. The shift toward trace-based performance analysis is supported by research demonstrating how detailed execution paths enable more accurate optimization efforts, especially in systems where performance variability can have critical consequences. This perspective positions tracing not as an ancillary monitoring concern, but as a core part of the engineering process. For CERN's Notifications System, integrating tracing into the rout-

ing pipeline allows the system to be understood, measured, and improved in a principled, data-driven way, aligning with both operational requirements and modern observability best practices.

3

Current Architecture - CERN Notifications

Contents

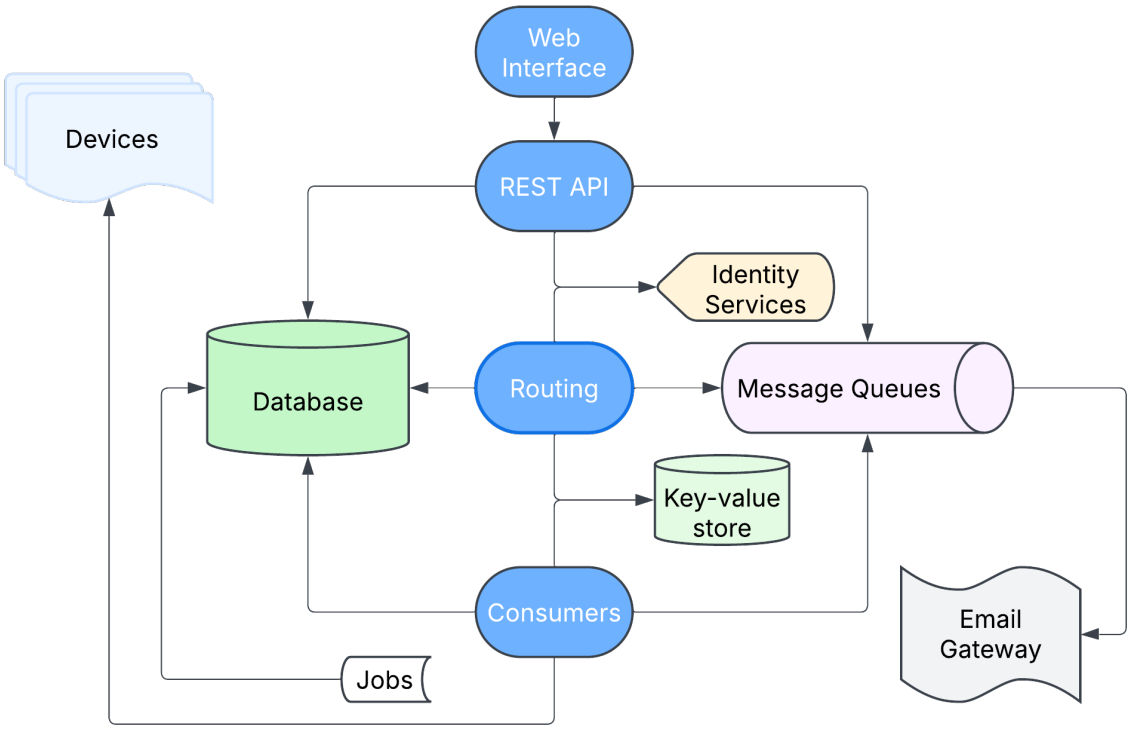

Figure 3.1: Existing Architecture

This chapter presents the architecture of the CERN Notifications system [24], outlining its core components, their roles, and how they interact to fulfill the service's objectives. Designed as a modular, event-driven system, it spans from the user-facing web interface to the backend logic, routing and delivery pipeline, and supporting services such as identity management, message queues, storage, and auditing. The architecture prioritizes reliability, scalability, and configurability, with clear separation of responsibilities across components. The following sections describe the high-level structure with the aid of a diagram, and then focus on the most relevant elements and design considerations.

3.1 Architecture Overview

The CERN Notifications service is built as a modular and decoupled platform for delivering targeted messages to users and systems. Its architecture aims for flexibility, maintainability, and scalability. The system is composed of several components that interact over a message-oriented middleware infrastructure. Figure 3.1 is an architecture overview.

At the user-facing layer, a web interface provides the main point of interaction, allowing users to interact with the service and perform a suite of actions such as: configure channels, their preferences

and devices, send notifications, etc. The web portal communicates with the backend via a RESTful API.

The backend server, illustrated as the ”REST API” component in the figure 3.1, serves an API that users and systems can consume. The backend is responsible for checking the authorization on the request and validating it before fulfilling it. It handles all interactions with the system's database when applying operations on channels, notifications, preferences, devices, etc. When a request to send a notification is received, the backend carries out any logical processing, audits any relevant changes, and adds a message to the router message queue.

The router is the component that carries out a very logic-intensive part of the processing pipeline. It is responsible for taking the notification message and its targets and applying multiple types of filtering logic. This includes expanding groups to users, resolving memberships, applying preferences, and taking mutes into account to arrive at the final user target list. The routing logic can transform a single notification into thousands of individualized messages, each of which is handed off for delivery by the dedicated consumer component via their respective message queues.

Each consumer component is tailored to a specific delivery medium such as email, SMS, push notifications, or chat platforms. The consumers retrieve messages from their queues, prepare the payload, and execute delivery to the respective endpoints.

Persistent state is stored in a PostgreSQL database, which is managed by CERN's storage infrastructure. It holds all information regarding user and channel definitions, user preferences and mutes, notifications, and other metadata required for the proper functioning of the system. An etcd-based key-value store is used both to maintain an auditable trace of operations and to provide deduplication functionality, ensuring that no operation is unintentionally repeated during error recovery.

At CERN, identity and group management is handled by a centralized identity service that maintains authoritative records for all accounts, their associated identities, and their group memberships. This service plays a fundamental role in enforcing access control and resolving group-based permissions across the infrastructure. It plays an essential role in determining which users belong to a given group or channel, and therefore, who should receive specific communications. As an external component, this identity service is accessed programmatically through a dedicated API, which enables system components, most notably the backend and routing, to query account and group information.

3.2 Architecture Considerations

While the overall system architecture is designed to be modular and scalable, specific performance considerations apply to each component. The backend server is built using Node.js [25] and benefits from its event-driven, non-blocking I/O model, allowing it to efficiently handle large volumes of concurrent requests. Most operations initiated at this level, such as database queries or queue insertions, are

lightweight and asynchronous, which keeps the response latency low even under significant load.

The consumer components, responsible for the final delivery of notifications to end devices, are designed in such a way that horizontal scaling is possible. Since each message item in their input queues is independent, multiple consumer instances can be deployed to parallelize the processing load. This is particularly effective in high-traffic situations, where increasing the number of workers directly improves throughput without introducing complexity or inconsistency.

In contrast, the Python-based routing component presents unique performance challenges. Although it also consumes from a queue, its processing workload is logic-intensive and may involve expanding group memberships into thousands of individual recipients, applying filtering rules, and making network requests. These operations are inherently stateful and tightly coupled. Horizontal scaling would help in dealing with situations with a high amount of notifications being sent. In the case that a notification is sent to a large or deeply nested group (i.e, one notification targeting a large portion of the community), the router becomes a bottleneck. As such, this component requires more targeted optimization strategies, which are explored in the following chapters.

To better understand the internal mechanics of the system and the expansion process involved, consider the flow of a notification targeting a group, as represented on 3.2. The backend server received a request to send a notification. It handles the request by validating the input, checking permissions, recording an audit entry, and making the relevant changes in the database. It then inserts a single message into the router message queue. This message is picked up by the router, it expands the group into its individual members. To do so, it queries the external identity service to resolve the group memberships and retrieve the relevant user accounts. For each user, the system determines the applicable delivery preferences and associated devices. This can lead to multiple delivery methods per user, such as email, SMS, or other supported channels, resulting in several distinct message objects per user. These messages are then individually enqueued in the appropriate queues corresponding to each consumer type. Each consumer processes and delivers the messages according to its specific mechanism. From a single input notification, this process can result in a substantial number of final delivery messages, highlighting the expansion in workload introduced by group-based targeting.

This architectural analysis reinforces the rationale for focusing optimization efforts on the router component, particularly in light of the broader goal of improving the system's responsiveness in critical communication scenarios. As established in the problem statement in Chapter 1, the main concern addressed in this work is performance, specifically, overcoming bottlenecks that may prevent the service from meeting the community's growing need for fast and reliable information delivery. Unlike the backend or consumer components, which benefit from straightforward scaling strategies, the router's logic-heavy operations and dynamic group resolution mechanisms make it more susceptible to delays. To enable a focused analysis of this component, a Jaeger all-in-one [26] container is deployed alongside

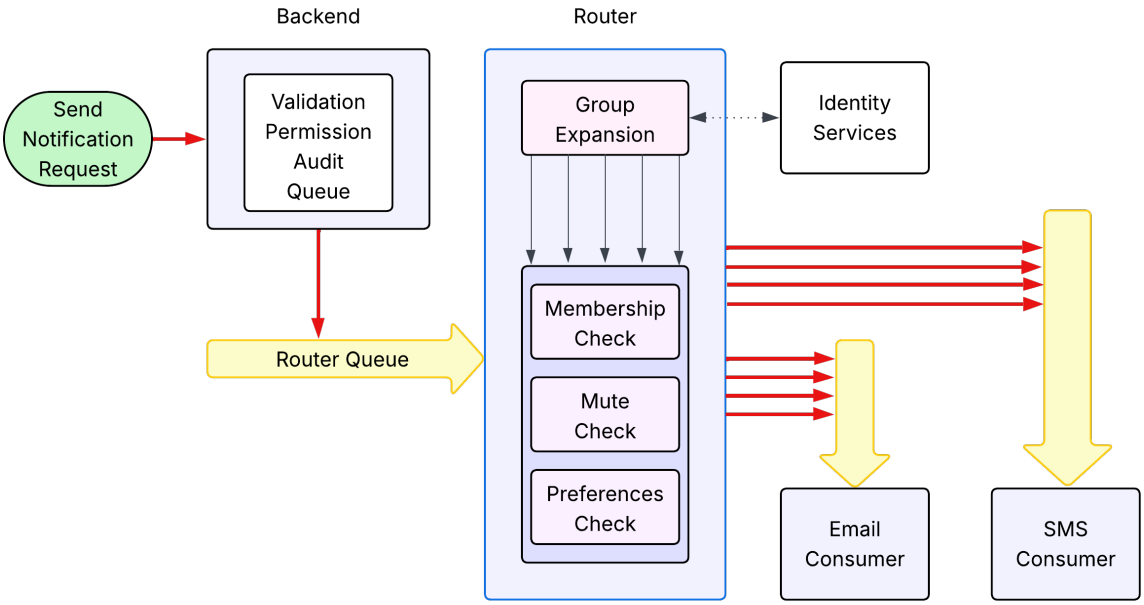

Figure 3.2: Notification Flow

it. The all-in-one configuration bundles the collector, query engine, UI, and storage backend, offering a self-contained environment that is especially suitable for instrumentation and diagnostics in development and evaluation stages. With this setup, it becomes possible to gather precise execution data from the router, supporting a trace-based investigation into possible issues and how those can be effectively addressed to improve the system's capacity to reach a large portion of CERN's personnel in as little time as possible.

4

Performance Assessment and Analysis

Contents

This section will outline the overall methodology adopted to profile the system. The selected test cases represent different types of effective workload for the system. It also addresses how the system is set up during the testing phase. An analysis of how the system responds to each test and possible problems that can be pinpointed.

4.1 Experimental Setup and Methodology

To support a consistent and isolated testing environment, the system was deployed on a dedicated machine running Ubuntu. This ensures minimal interference from external processes and provides a reproducible baseline for performance analysis. The router component, which is the focus of the performance investigation, is deployed alongside a Jaeger all-in-one instance. Jaeger serves as the distributed tracing infrastructure, allowing detailed insight into the internal operation of the router through trace spans and timing breakdowns. For the router to function correctly and reflect production-like behavior, supporting components such as the database and etcd key-value store are also deployed locally. This setup mirrors the key architectural elements that are relevant for the test conditions.

Before describing the performance tests, it is useful to clarify the key concepts used across the system and in the analysis. The system defines several core abstractions:

- User - Represents an individual identity.
- Group - A collection of users or groups.
- Device - An endpoint associated with a user through which notifications can be delivered.
- Preference - Set by the user regarding when and which devices to be used as the delivery endpoint.
- Mute - Set by the user regarding whether or not notifications from a certain channel should be received during a certain period.
- Channel - A channel is a grouping construct. It is through a channel that a notification is sent. That notification aims to be delivered to that channel's members through some form or another. A channel is managed by one or more users.
- Notification - Fundamental object transmitted through the system, which encapsulates content and targeting information.

In addition to these core constructs, the system supports a more advanced targeting mechanism based on intersections. In certain scenarios, a notification is not intended for the entire membership of a channel but only for those who are members of both the channel and an explicitly defined group. This allows for precise targeting within a large or broadly defined channel, ensuring that only users meeting

specific criteria receive a given message. Implementing this form of conditional targeting adds another layer of complexity to the routing process, as the system must efficiently compute the intersection between potentially large and deeply nested group structures during the message expansion phase.

The testing methodology is built around controlled workload scenarios designed to stress the router logic in different manners. Tests are carried out by initiating notification delivery requests either through the front-end interface or directly via API to the backend, both of which result in the insertion of a message into the router message queue. Variations in the test inputs include the number of users and/or groups that are members of the channel, and the characteristics user for filtering or matching. These dimensions are chosen to surface the performance limits of the router under diverse operational conditions. Each test scenario corresponds to a specific combination of these variables. A summary table that illustrates the variety of the test scenarios is provided in Table 4.1.

Test	Users	Groups	Intersection
1	multiple	1	No
2	multiple	multiple	No
3	multiple	0	Yes - large group
4	multiple	0	Yes - small group
5	0	1 - large	Yes - large group
6	multiple	0	Yes - large group

Table 4.1: Test Cases

4.2 Performance Baseline and Timing Results

This section presents the empirical performance baseline of the system as observed through a series of benchmark test cases, each designed to simulate realistic notification delivery scenarios with varying user and group configurations. The goal is to establish a quantitative understanding of runtime behavior under different conditions, including direct user targeting, group-based resolution, and combinations thereof. Subsection 4.2.1 provides an overview of all test cases, highlighting their structural characteristics and summarizing observed execution times. Subsequent subsections (4.2.2 through 4.2.7) provide a detailed breakdown of each test case individually, analyzing control flow, performance implications, and points of inefficiency that motivated later optimization designs.

4.2.1 Total Execution Time

To contextualize the performance of the current system implementation, this section presents the total execution time observed for each of the defined test cases. The execution time measured corresponds to the duration taken by the router from the reception of a notification to the point where all targeted users have been processed for delivery. The results are shown in Table 4.2.

Table 4.2: Total execution time per test case

Test Case	Total Execution Time
1	1.89 s
2	5.86 s
3	45.10 s
4	2.64 s
5	207.00 s
6	206.00 s

The total execution times across the different tests can already provide some clues as to what might influence performance. Test cases 1 and 2, which use direct user and group membership, result in relatively low execution times, increasing for test 2, consistent with the greater number of groups and users, implying that execution time increases with the number of distinct entities being resolved.

Test case 3 introduces intersection targeting with a large group and results in a sharp rise in execution time. This suggests performance might be hindered by intersection logic when applied to a large user base. Test case 4 also uses intersection but with a much smaller target group; however, it has a much shorter execution time, supporting the conclusion that the group's size, rather than the intersection logic itself, plays a more impactful role in execution time.

Test cases 5 and 6 show the highest execution times. Both involve targeting a large group, but differ in their channel composition: test case 5 includes a large member-group, whereas test 6 includes many member-users. The nearly identical execution times suggest that the scale of the group could dominate the performance cost, more than how the channel is structured.

This preliminary look at execution times in relation to test characteristics offers valuable insight for guiding the more detailed performance analysis that follows.

To gain deeper insight into the causes behind the observed execution times, we now turn to a closer examination of the tracing data collected during each test.

4.2.2 Test case 1 - multiple users

When analyzing the trace execution for this test, there is a segment that stands out and dominates the overall elapsed time. The portion regarding fetching and expanding any involved groups into their members. Figure 4.1 shows a part of the tracing visualized on what can be called a span tree, regarding that particular section.

Figure 4.1: Test 1 - Getting group

Following the most lengthy spans, it is plain to see the most time-consuming section to be an HTTP POST request. This request is made to the CERN Auth Service, and corresponds to approximately 73% of total execution time.

Another section of note is further along, where a sequence of spans displaying a section where a list of retrieved users is iterated over to perform other actions. It is illustrated in Figure 4.2.

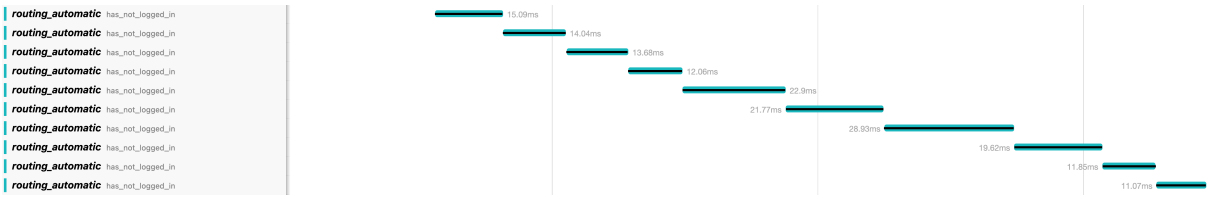

Figure 4.2: Test 1 - Iterating Users

4.2.3 Test case 2 - multiple users and groups

This test differs from test 1 by introducing multiple groups, causing the time to grow linearly with the number of groups.

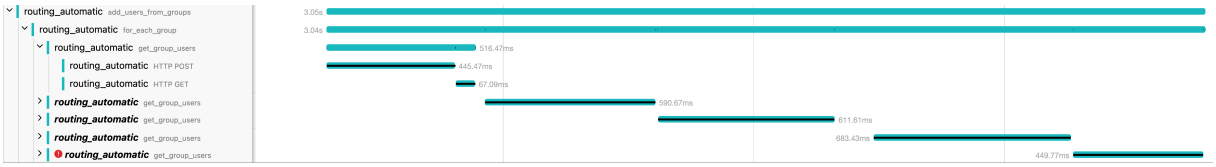

Figure 4.3: Test 2 - Getting Groups

For each of the groups, the system must also iterate through each of the members that it has expanded to and perform queries on them to build an appropriate user object.

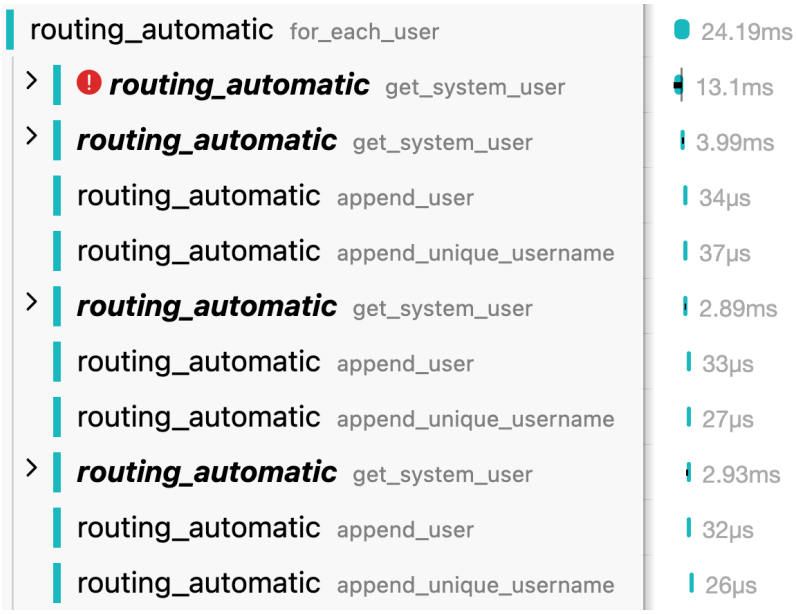

Figure 4.4: Test 2 - Building Users

The final section of note is iterating over the final user list and performing checks for each of the users regarding mutes, preferences, and delivery methods.

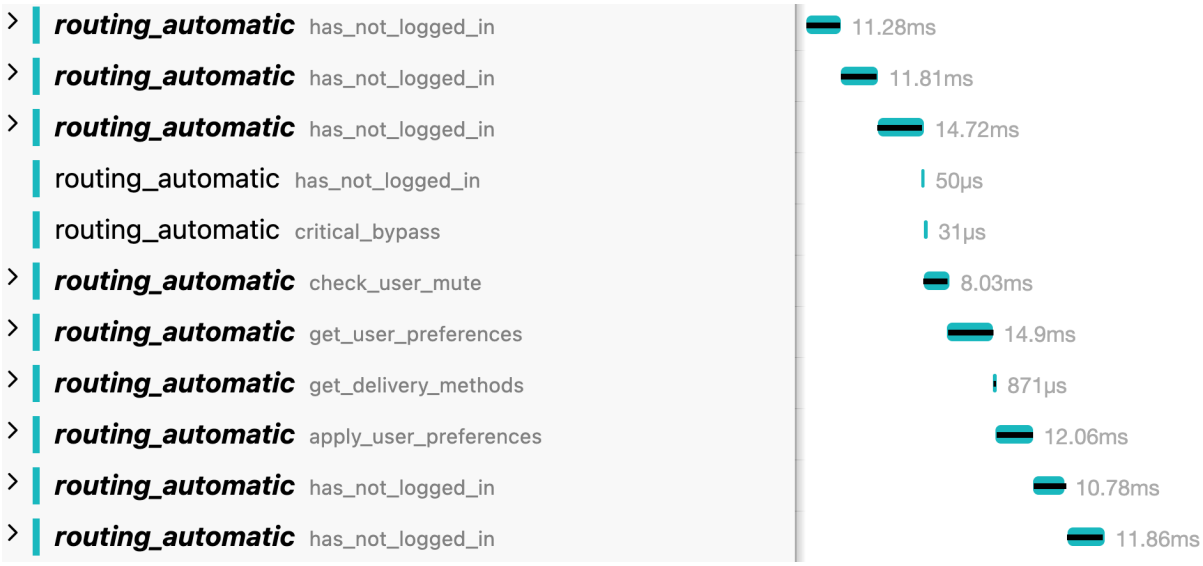

Figure 4.5: Final User list check

It is already easy to draw as a preliminary conclusion that even slightly scaling the number of groups or users can have a meaningful impact on total execution times.

4.2.4 Test case 3 - multiple users intersect large group

This test case is the first one to make use of the intersection capabilities of the system. This trace confirms yet again that a substantial portion of execution time is spent in resolving group members via CERN's authorization service API. Several consecutive HTTP GET spans accessing `https://authorization-service-api.web.cern.ch/`. Each request spans from 600 ms to over 1100 ms, repeated multiple times as the group's user list is returned in paginated slices, illustrated in Figure 4.6. This results in a cumulative delay of several seconds, emphasizing the cost of external group resolution for large groups.

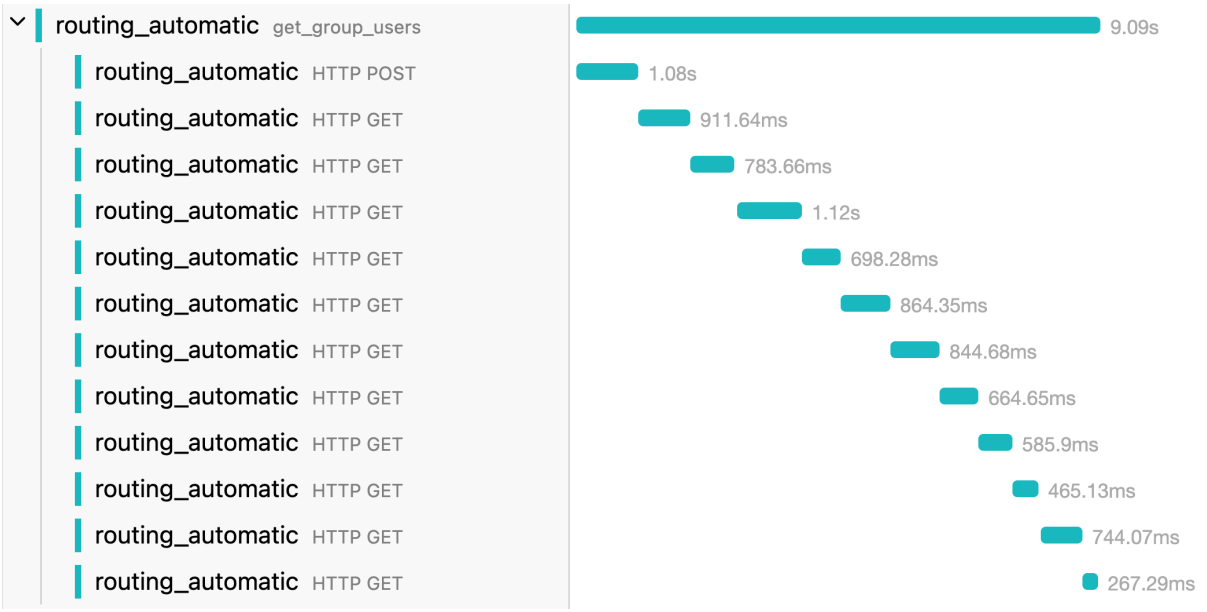

Figure 4.6: Test 3 - Paginated Group Return

Right after that, a series of database operations occur where user-level lookups are performed, iterating over all of the previously expanded users. While individual queries are fast (in the 1-3 ms range), their accumulated cost is non-trivial due to the large number of users, adding up to 34 s. The remainder of the execution time is spent on gathering channel groups and users, which is negligible since, in this test case, there are no member groups, the targeted intersection is calculated, and then the same iteration checking and applying preferences and mutes for each user before ending.

4.2.5 Test case 4 - multiple users intersect small group

Test case 4 features only a small group, hence a much smaller execution time. Yet the `get_target_users_router` span lasted 1.42 s, which is still over half the total run time for this test case 4.7. Group resolution still emerges as a significant contributor to overall latency. In this span,

`add_users_from_groups` specifically consumed approximately 947 ms, clearly indicating that resolving group memberships represents a substantial performance bottleneck.

Throughout the rest of the execution, the system conducts multiple database queries to validate channel subscriptions, user preferences, and mute statuses.

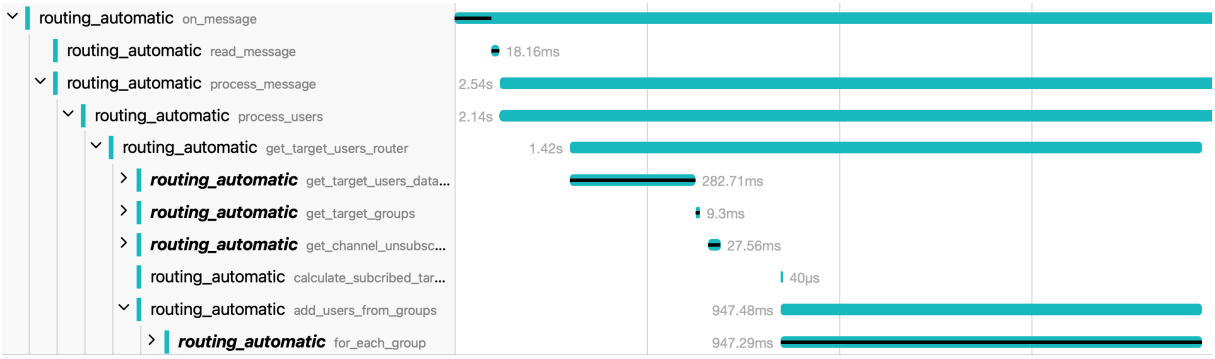

Figure 4.7: Test 4 - Starting Segment

This trace analysis already reveals a predominantly sequential execution model where critical path operations, including group resolution, preference validation, and delivery, are executed in series.

4.2.6 Test case 5 - large group intersect large group

Test Case 5 is the first of the two high-latency benchmark scenarios and was constructed to simulate a stress condition combining both a large membership group and an equally large intersection target group. This setup is representative of a real-world situation where high-volume notifications are scoped to intersecting organizational units. This could be a common occurrence when the groups in question have high overlap, which would be highly likely when using the most encompassing groups to reach as much of the personnel as possible.

The trace begins with the group resolution phase, which, as in earlier tests, emerges as a dominant source of delay. The group queried contains a substantial number of users, triggering the CERN Authorization Service to return the data in paginated form. Each paginated HTTP request introduces a blocking delay on the order of several hundred milliseconds, and with multiple pages to fetch, the cumulative latency of this phase alone reaches several seconds, as can be seen in Figure 4.8.

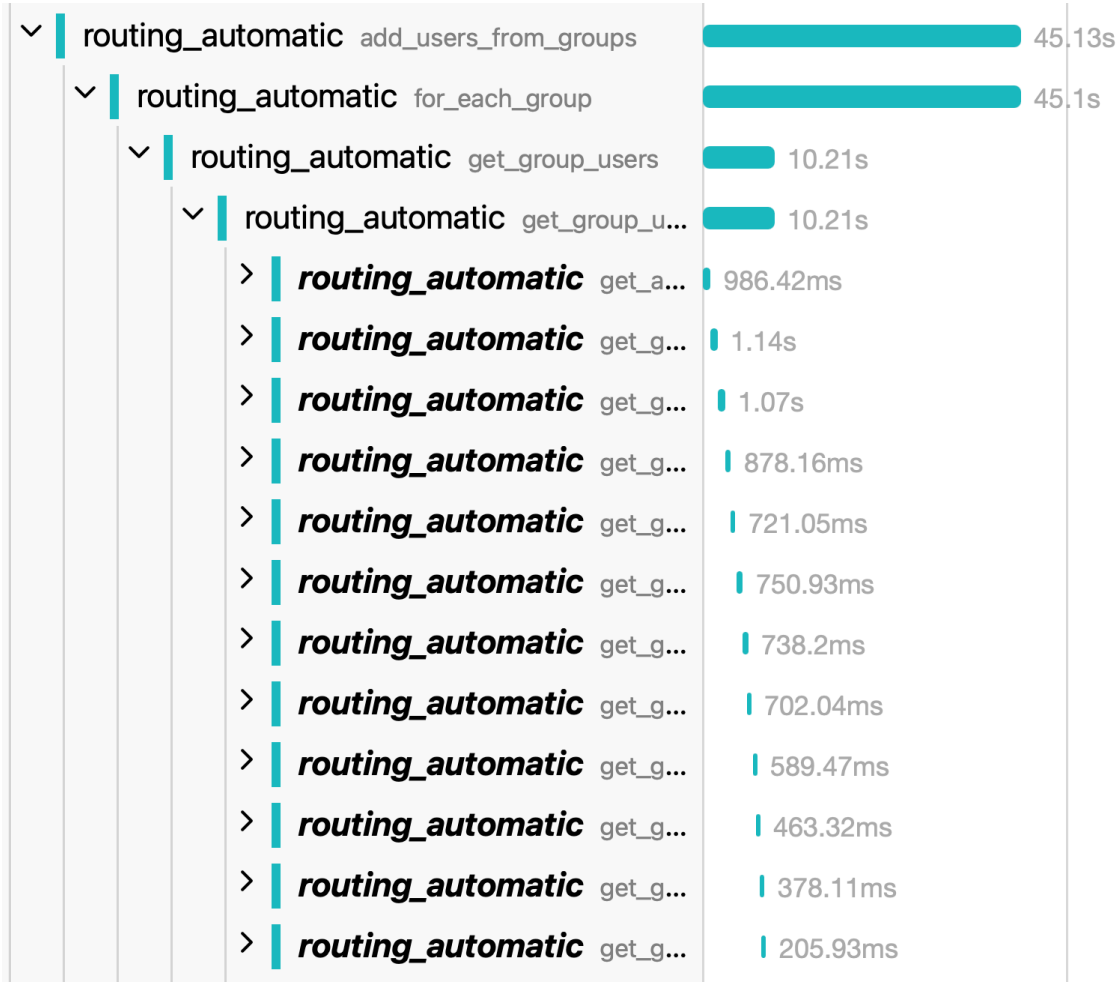

Figure 4.8: Test 5 - Resolving Group Users

Notably, the trace shows these requests are executed sequentially, resulting in serialized wait times. This behavior confirms prior findings that the group resolution mechanism is a systemic bottleneck.

Once the full set of group members has been retrieved, the router proceeds to the intersection logic. In this test, both the channel group and the target group contain large and largely overlapping user bases. The output of the intersection step yields a high cardinality span, thousands of users, which then become the input to a more resource-intensive per-user processing phase.

The trace clearly delineates this next phase with a repeating pattern of spans corresponding to individual user handling. For each user, the system performs several steps: preparing the user object, validating settings, fetching supplementary data from the database, and appending the result to the output structure. Each of these steps, although lightweight in isolation, becomes a major contributor to total execution time when multiplied over thousands of users (Figure 4.9).

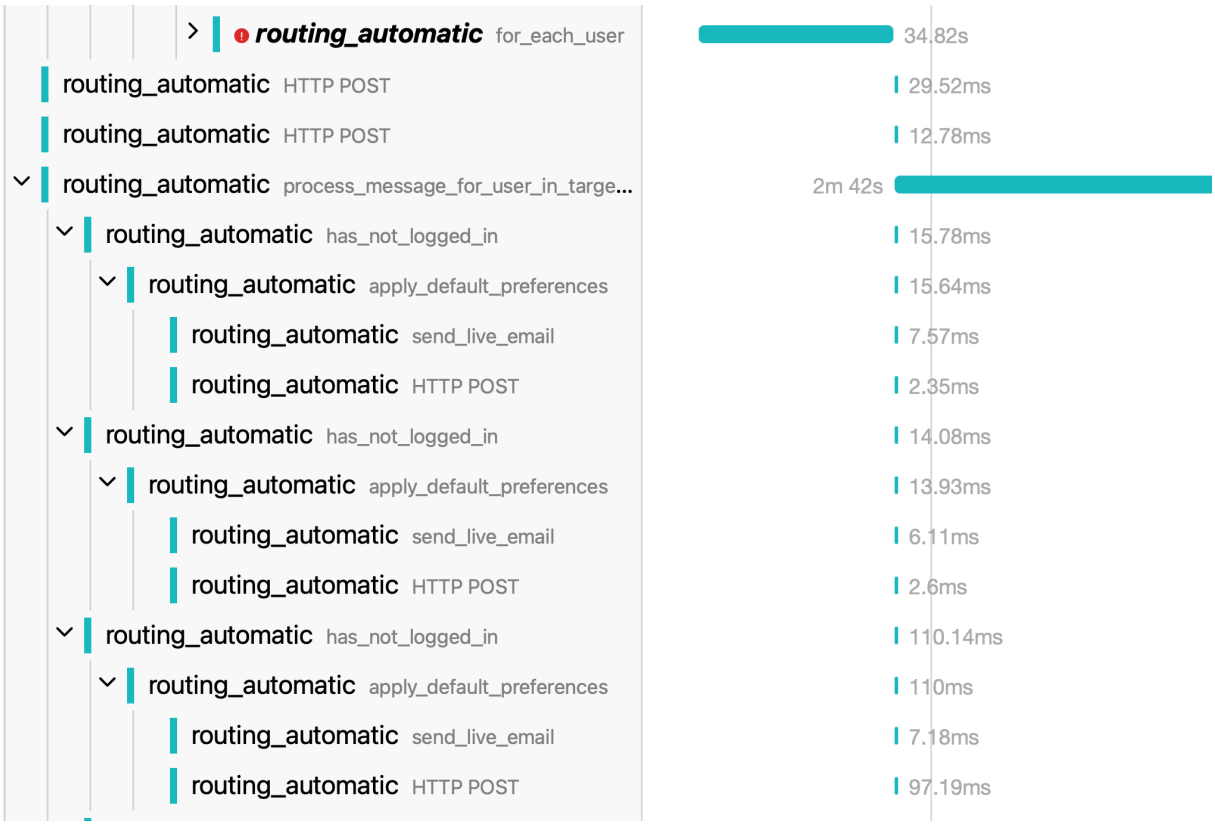

Figure 4.9: Test 5 - Dispatching Messages per User

4.2.7 Test case 6 - multiple users intersect large group

This test case reflects a scenario highly similar to the previous test, where both the number of group members and direct user members are high. This test, however, replaces the large member group with a large list of direct member users, while still targeting a large group. The total execution time is approximately the same. Just as in test 5, group resolution through CERN's Authorization Service emerges again as the primary bottleneck. The trace once again shows the iterative user processing following the group resolution. Each user is processed individually. Despite reducing the initial resolution pressure seen in test case 5 by avoiding a large member group, total execution time is equivalent, driven by the sheer number of users targeted and per-user operations. Figure 4.10 displays the overview of test case 6 execution.

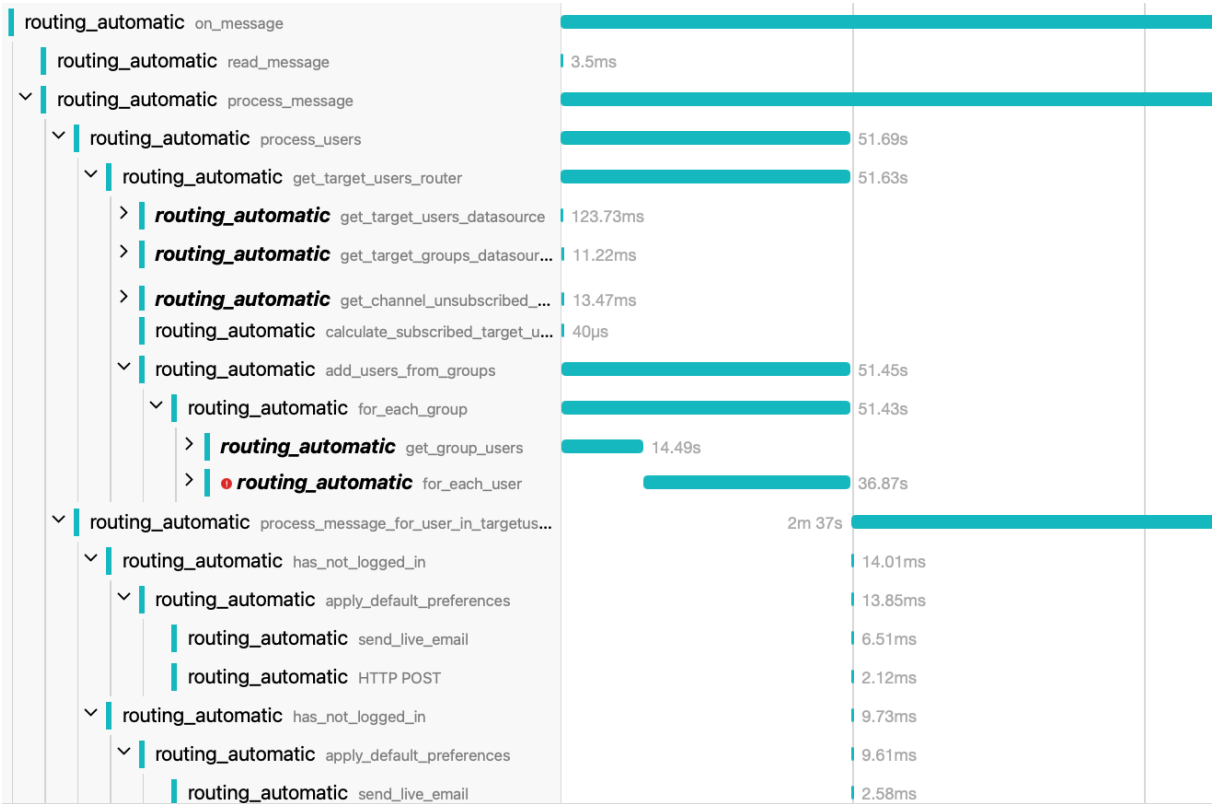

Figure 4.10: Test 6 execution - overall view

4.3 Performance Analysis and Solution Proposals

The series of controlled test scenarios and their corresponding trace analysis provide an insightful view of the current performance profile of the CERN Notifications system, particularly the routing component. Each test case, constructed to incrementally increase complexity and stress specific parts of the system, has revealed a consistent set of bottlenecks and architectural patterns that underlie current limitations in scalability and latency.

4.3.1 Inefficient Sequential Processing

A key contributor to poor performance in large-scale scenarios (such as test cases 3, 5, and 6) is the system's reliance on a strictly linear execution model for handling both user and group-level operations. This results in two intertwined inefficiencies: a high volume of small, repetitive tasks for each user and group, and the absence of any asynchronous or parallel execution strategy to mitigate their cumulative cost.

Listing 4.1: router.py - get_channel_subscribed_users - Pseudo-code

```
function get_channel_subscribed_users(channel_id):
    channel = fetch_channel(channel_id)

    unsubscribed_ids = []
    for user in channel.unsubscribed:
        unsubscribed_ids.append(user.id)

    subscribed_users = []
    for member in channel.members:
        if member.id not in unsubscribed_ids:
            user = build_user(member)
            subscribed_users.append(user)

    return subscribed_users
```

Many of the routing component's core tasks, such as validating users, checking for mutes, determining device delivery preferences, and expanding group memberships, are performed in a strictly serial fashion. As a result, the total processing time increases linearly with the number of users and groups involved. While each individual operation may be computationally inexpensive (often taking only tens to hundreds of microseconds), they must be repeated thousands of times per notifications, leading to a significant aggregate delay.

Listing 4.2: authorization_service.py - get_group_users_api - Pseudo-code

```
function get_group_users_api(group_id):
    token = get_auth_token()
    data = []

    response = request_group_members(group_id, token)
```

```
    data.extend(response.members)

    while response.has_next_page:
        response = request_group_members(group_id, token, next_page=response.next)
        data.extend(response.members)

    group_users = []
    for member in data:
        user = prepare_user(member)
        if user:
            group_users.append(user)

    return group_users
```

This behavior is particularly evident in operations like fetching subscribed users to a channel (Listing 4.1) and resolving group membership from external APIs (Listing 4.2). In both cases, iterative loops are used to process each user sequentially, even though each iteration is independent and could be safely executed in parallel.

Listing 4.3: router.py - get_target_users - Pseudo-code

```
function get_target_users(notification_id, channel_id):
    target_users = fetch_target_users(notification_id)
    target_groups = fetch_target_groups(notification_id)

    if target_users and not target_groups:
        return filter_unsubscribed_users(target_users, channel_id)

    unsubscribed_users = get_unsubscribed_users(channel_id)
    subscribed_target_users = []

    for user in target_users:
        if user.username not in unsubscribed_users:
            subscribed_target_users.append(user)

    if target_groups is not empty:
        add_users_from_groups(notification_id, channel_id,
        subscribed_target_users, target_groups, unsubscribed_users)

```

```
19      return subscribed_target_users
```

The situation is further exacerbated in group targeting logic (Listing 4.3 and Listing 4.4), where group expansion and user resolution are both performed one group or user at a time. This leads to patterns where large delays accumulate simply because tasks are waiting for previous, unrelated tasks to complete. For example, group memberships are resolved in order, with no concurrent fetches even when all group IDs are known ahead of time. The inner loop that processes each group's users is similarly serial, requiring individual database queries for every new user.

Listing 4.4: router.py - add_users_from_groups - Pseudo-code

```
1  function add_users_from_groups(notification_id, channel_id,
2      users, groups, unsubscribed_users):
3      known_usernames = [user.username for user in users]
4
5      for group_id in groups:
6          group_users = get_group_users(group_id)
7
8          for user in group_users:
9              if user.username in known_usernames:
10                 continue
11             if user.username in unsubscribed_users:
12                 continue
13
14             system_user = get_system_user(user.username)
15             if system_user exists:
16                 users.append(system_user)
17             else:
18                 users.append(user)
19
20             known_usernames.append(user.username)
```

This design choice severely limits scalability. As the user base grows or messages target larger audiences, routing execution time increases proportionally, something that becomes highly problematic in time-sensitive or high-throughput contexts.

In summary, the root inefficiency lies not just in the volume of operations but in the lack of mechanisms to speed them up, such as parallelism for independent tasks. The observed patterns in Listings 4.1 through 4.4 make clear that the system fails to exploit opportunities for concurrency, resulting in deterministic but inefficient runtime behavior that scales poorly under load.

Optimization Proposal: Addressing Parallelism

A recurring inefficiency identified throughout the routing component is the use of strictly sequential iteration for operations that are independent and safe to execute in parallel. This pattern appears most notably in per-user processing, such as applying preferences, resolving devices, and checking mutes, but also in other workflows, such as resolving multiple groups in the same routing pass. In these cases, each operation is disjoint from the other, does not depend on shared mutable state, and is typically I/O-bound (e.g., involving external API or database calls).

Sequential execution of these operations, even if each is individually fast, results in linearly increasing latency under scale. This behavior was observed across multiple test cases where thousands of users and groups are involved, and the time required to process them grows proportionally.

To address this, selected sections of the routing logic were refactored to use parallel execution models. Specifically, Python's `ThreadPoolExecutor` was introduced to handle collections of independent, iterable tasks, allowing multiple items (e.g., groups, users) to be processed concurrently. Examples of these changes can be seen in Listing 4.5. This approach is particularly suitable for I/O-bound tasks, which benefit from being issued simultaneously due to their latency profile. One such example is the group resolution step in `add_users_from_groups`, where each group's membership can be resolved independently of others. The same logic is also applied to other instances of iterative execution, such as processing user preferences, checking user devices, mutes.

When it comes to computational complexity, there are a few nuances to be stated. Before parallelization, operations executed sequentially over N elements (users, groups, etc.) typically have a time complexity of $\mathcal{O}(N \cdot T)$, where T is the time taken per task (e.g., HTTP call, DB query). After parallelization, the effective wall-clock complexity becomes $\mathcal{O}(max(T))$, assuming tasks are independent and I/O-bound with ideal conditions (no contention, enough threads/resources). This of course represents a best-case scenario and not a true algorithmic complexity shift, but a practical runtime gain.

Multithreading is not without its setbacks. When opting for multithreading, one must account for task submission overhead; creating and managing threads, especially if very small tasks are submitted, can offset the gains. Overhead grows with the number of threads and context switches. Using a fixed thread pool helps mitigate this, a number that should be revised and adjusted to best suit the needs of the system in production. There is also the matter of server-side throttling, which might occur if the auth service itself can't handle concurrency well, parallelism could cause saturation. The actual gain is bound by the responsiveness and scalability of external services. In practice, these adjustments allow the routing component to handle large fan-out notification events more efficiently, especially under high concurrency conditions typical of critical alerting scenarios.

Listing 4.5: Parallelize Iterative Work - Pseudo-code

```
function add_users_from_groups():
    // Fetch users from multiple groups in parallel
    with thread_pool:
        futures = submit_all(get_group_users(group_id) for group_id in groups)
        group_users = set()

        for future in completed(futures):
            result = future.result()
            usernames = extract_usernames(result)
            group_users.update(usernames)

    // Filter unsubscribed users
    unique_usernames.update(group_users)
    unique_usernames.remove_all(unsubscribed_users)

    // Fetch full user data in parallel
    with thread_pool:
        futures = submit_all(get_system_user(username) for username in unique_usernames)
        temp_users = []

        for future in completed(futures):
            result = future.result()
            temp_users.append(result)
```

4.3.2 Membership Testing

A fundamental inefficiency in the current implementation arises from the exclusive reliance on the list data structure for tasks where other, better suited structures would lead to better performance and readability. This design decision impact both runtime performance and architectural extensibility, particularly in scenarios involving high user volume or multithreaded execution models.

Membership test (x in list) is inherently linear in complexity, as each lookup may require scanning the entire list. This becomes problematic when such lookups are performed repeatedly inside nested loops or user/group resolution functions. A representative case can be seen in this snippet from `get_target_users` function in Listing 4.6.

Listing 4.6: router.py - Membership testing example snippet- Pseudo-code

```
for user in target_users:
    if user.username not in unsubscribed_users:
        subscribed_target_users.append(user)
```

Here, `unsubscribed_users` is a list, and for every user in `target_users`, a full scan of that list is performed to test membership. In workloads where `target_users` and `unsubscribed_users` each contain hundreds or thousands of entries, this results in a quadratic number of comparisons, significantly increasing latency.

Another case of this pattern appears in the `get_channel_subscribed_users` function, for which a snippet is presented in Listing 4.7.

Listing 4.7: router.py - Membership testing example snippet 2 - Pseudo-code

```
unsubscribed_ids = [user.id for user in channel.unsubscribed]
return [
    self.__build_user(member) for member in channel.members
    if member.id not in unsubscribed_ids
]
```

Again, the membership test (`member.id not is unsubscribed_ids`) uses a list for `unsubscribed_ids`. While the code is functionally correct, the performance cost scales with the size of the list and the number of members, with each lookup taking $\mathcal{O}(N)$ time.

These performance issues are not theoretical: they manifest concretely in the system's latency profiles under load. Moreover, lists are fundamentally ill-suited for concurrent access and mutation, which makes them a poor fit for any future attempts to introduce multithreading or parallel execution in the routing logic. Since a list is not thread-safe and has no inherent concurrency controls, sharing or modifying them across threads requires locking, which would further degrade performance and increase complexity.

Replacing lists with set structures, where appropriate, enables constant-time membership checks. This change would be beneficial in many routing paths due to repeated checks on large collections. It is also a low-complexity optimization that can yield both immediate and long-term benefits.

Optimization Proposal: Leveraging Sets

Lists are ubiquitously used in the routing component to represent collections of users, groups, and other objects. While lists are simple to use, they introduce some noteworthy limitations.

To address these inefficiencies, the improvements aimed to replace many uses of `list` with `set` or

`frozenset`, depending on the context. This change was motivated by both performance considerations and structural clarity. One of the core benefits of sets is their support for constant-time membership checks. Unlike lists, which must scan each element to determine whether a value exists, leading to a worst-case time complexity of $\mathcal{O}(N)$, sets leverage hash-based indexing, allowing such a check to complete in $\mathcal{O}(1)$ on average. This difference becomes critical in performance-sensitive sections of the routing logic where membership is checked repeatedly across potentially large collections of users or identifiers.

Moreover, the types of data commonly handled by the routing component, such as usernames, user IDs, and group names, are all inherently unique, string-based identifiers. These identifiers are not only uniquely assigned, but also stable and hashable, making them an ideal fit for set-based operations. There is little to no practical risk of hash collisions, and no need for ordering, which further reinforces the suitability of sets over lists for this purpose.

Beyond performance, sets also enforce semantic correctness. Since each user or group should logically appear only once in any working set of recipients or targets, the automatic deduplication behavior of sets helps to prevent unintended duplications that could otherwise arise when combining or expanding multiple lists. This is particularly useful during group resolution, where users may be members of several groups, and care must be taken not to produce redundant delivery actions.

In scenarios involving parallel execution, such as when group membership or user preferences are resolved concurrently, sets also offer structural advantages. `frozenset`, in particular, is an immutable variant that can be safely passed between threads or used in cached contexts without risk of modification. This is crucial in maintaining thread safety and ensuring that concurrent operations do not introduce race conditions or inconsistencies.

Finally, sets also simplify operations that are semantically aligned with the system's needs. Tasks such as intersecting the members of two groups or excluding unsubscribed users from a pool of targets are not only more efficient when done with sets, but also more readable and maintainable, as they directly express the logic of the operation.

A prototype proposal modification is shown in Listing 4.8. It replaces the `list`-based collections with `set` structures. Most notably, the use of the
`difference_update` method allows the system to subtract unsubscribed users from the target set in a single pass. The runtime complexity of `set.difference_update()` on a set with n elements and a second set with m elements is $\mathcal{O}(N)$ because each element in the first set must be checked for membership in the second. Given that set membership testing is $\mathcal{O}(1)$, the resulting complexity is $\mathcal{O}(N) \cdot \mathcal{O}(1) = \mathcal{O}(N)$. However, when the second set is smaller ($M < N$), the actual cost can be considered $\mathcal{O}(M)$, making this operation highly efficient for common notification scenarios involving a minority of unsubscribed users.

Listing 4.8: Set and Difference Update prototype solution proposal - Pseudo-code

```
function get_target_users(notification_id, channel_id):
    target_users = set(get_target_users_from_db(notification_id))
    target_groups = set(get_target_groups(notification_id))

    if not target_users and not target_groups:
        return empty_set

    unsubscribed_users = set(get_unsubscribed_users(channel_id))

    if target_groups:
        add_users_from_groups(notification_id, channel_id, target_users,
            target_groups, unsubscribed_users)

    target_users.difference_update(unsubscribed_users)
    return target_users
```

This pattern avoids redundant iteration and scales more predictably with user and group volume. Moreover, it sets a solid foundation for later concurrency improvements, as `set` operations are more naturally decomposable and thread-friendly than iterative scanning.

In a multithreaded environment, it's often desirable to ensure that shared data structures are not accidentally mutated during execution. Python's `frozenset` provides an immutable version of the built-in `set`, which can be safely passed between threads since its contents cannot be changed.

This makes `frozenset` especially useful when a set of items, such as usernames, identifiers, or user objects, is read frequently but never modified. It ensures defensive immutability and avoids race conditions caused by unintended mutations.

A typical usage scenario is shown in the Listing 4.9, where a list of users is converted to a `frozenset` before being submitted to multiple threads for concurrent processing.

Listing 4.9: Frozenset prototype solution proposal - Pseudo-code

```
function parallel_check_settings_and_send(message, users):
    # Convert the user list to an immutable frozenset
    users = frozenset(users)

    # Create a thread pool executor
    with ThreadPoolExecutor:
        futures = []
```

```
        for user in users:
            # Submit each user processing task to the executor
            futures.append(executor.submit(check_settings_and_send, user, message))

    # Collect results and handle any exceptions
    for future in as_completed(futures):
        try:
            result = future.result()
        except Exception:
            log("An exception occurred during user message dispatch.")
```

4.3.3 Dominant Bottleneck - Group Resolution

Across nearly all test cases involving group-based delivery, specifically test cases 3, 5, and 6, a single dominant bottleneck was observed: the resolution of group memberships via the CERN Authorization Service. This process is implemented using synchronous HTTP GET requests to an external API endpoint, which returns group members in a paginated format. The implementation is sequential and blocking; each page of results is fetched and processed fully before the next request is made. When groups are large, this results in a cumulative and significant delay for each group processed.

This bottleneck is illustrated in the pseudo-code in Listing 4.1, which represents the underlying logic for resolving group members from the CERN Auth API. After obtaining an authorization token, successive calls to fetch members are made. These members are retrieved page-by-page using the pagination pointer included in the API response. As shown in the listing, each group is processed independently and serially, and all API interactions occur in a blocking fashion with no concurrency in place.

This implementation design reveals a key limitation: the responsiveness of the system is tightly coupled to the performance and availability of an external service. This dependency poses a substantial scalability challenge. In time-sensitive scenarios, such as the dissemination of critical alerts, the sequential, I/O-bound nature of this group resolution step becomes a serious performance constraint, preventing the system from meeting low-latency delivery targets.

Optimization Proposal: Caching and Concurrency

Given that external group resolution is consistently the dominant contributor to routing latency, particularly in scenarios involving large or multiple groups, it is of strategic importance to minimize the time spent on this operation. Since the latency originates outside the control of the system (i.e., within an external identity service), the most effective way to reduce this overhead is to avoid making the request altogether when possible.

A complete duplicate of CERN Authorization Service's group database is near impossible due to its sheer size and complexity, not to mention synchronization issues. An in-between solution can be achieved by leveraging caching mechanisms that store the result of previous group resolution calls. When a group has already been resolved recently, and no change in its membership is expected or critical, the cached data can be reused. This allows the system to bypass the repeated execution of high-latency API calls, significantly improving overall performance in both average and worst-case scenarios.

The implementation makes use of Python's `functools.lru_cache`, which is efficient, automatically handles memoization, and is thread-safe. The previous implementation and lookup for a group is done through a group ID, which provides a direct key to be used for caching. Listing 4.10 shows a simplified view of the cached group resolution proposal. This memoization implementation encapsulates the whole group return, meaning that the full paginated group membership is fetched and combined before caching, and the API behavior remains unchanged. The choice of cache expiration is at this time space-based (`maxsize=None`), which serves its purpose within a controlled deployment where expected cache usage is known. An important detail in the implementation is the use of a `frozenset` as the return type for the cached group members. This choice offers multiple benefits: it ensures immutability, preventing any accidental modification of the cached data after retrieval, it guarantees hashability, which is a requirement not met by `list`, and needed to be compatible with this type of caching.

Listing 4.10: Cached prototype solution proposal - Pseudo-code

```
from functools import lru_cache

@lru_cache(maxsize=None)
function get_group_users_api(group_id):
    token = get_access_token()
    headers = {"Authorization": "Bearer <token>"}

    all_members = []
    response = get_group_members(group_id, headers)
    all_members.extend(response.data)

    while response.has_next_page:
        response = get_group_members(group_id, headers, page=response.next)
        all_members.extend(response.data)

    return frozenset(
```

```
17          prepare_user(member)
18          for member in all_members
19          if prepare_user(member) is not null
20      )
```

While these changes already address some of the performance concerns around group resolution, namely via caching and concurrent group-level expansion, a proposal for this critical issue can go further by combining previously introduced optimization strategies can yield even further improvements, especially combined with an adjustment enabled by the analysis of the CERN Authorization Service's API documentation. This includes applying caching, multithreading, and set usage in tandem, and extending parallelism not just to group-level operation, but also within the process of fetching paginated membership data from CERN's external authorization service.

An analysis of the CERN Authorization Service's API documentation revealed that the endpoint previously in use for group resolution `/api/v1.0/Group/memberidentities/precomputed` had been deprecated. To address this, the implementation was updated to use a more modern and fully supported endpoint: `/api/v1.0/Group/{id}/members/identities/recursive`. This endpoint's return also includes pagination metadata such as:

```
1 "pagination": {
2   "total": 4312,
3   "offset": 0,
4   "limit": 100,
5   ...
6 }
```

With this information, the client can calculate the total number of result pages and use the `offset` parameter to fetch any page explicitly. The improved implementation takes advantage of this by:

1. Fetching the first page of results to extract the `total`, `limit`, `offset`.
2. Computing the remaining pages that need to be retrieved.
3. Dispatching concurrent HTTP requests for these pages.

Listing 4.11 shows a prototype pseudo-code proposal for the algorithm at a high level.

Listing 4.11: Reworked Group Resolution - Pseudo-code

```
1 function get_group_users(group_id):
2     # Fetch first page
```

```
    first_page = fetch_page(group_id, offset=0)
    data = parse_users(first_page)
    total = first_page.pagination.total
    limit = first_page.pagination.limit

    num_pages = ceil(total / limit)
    urls = [build_url(group_id, offset=i*limit) for i in range(1, num_pages)]

    # Fetch remaining pages concurrently
    with thread_pool:
        for each url in urls:
            launch fetch_page(group_id, offset=...)

    for each result in completed_futures:
        data.extend(prepare_user(result))

    return deduplicated_set(data)
```

Each page is processed independently, and each user object is constructed via the same `prepare_user()` logic used in previous implementations.

This method improves upon prior efforts by parallelizing deeper within the group resolution stack and reducing the latency profile from $\mathcal{O}(P * T)$ to closer to $\mathcal{O}(T)$, where P is the number of pages and T is the time to retrieve one page. It is important to emphasize that, once again, this optimization primarily affects the practical execution time in a multithreaded, I/O-bound setting, rather than changing the classical algorithmic complexity. The theoretical number of operations remains the same, and thus the asymptotic complexity class is unchanged. However, by overlapping I/O-bound tasks across multiple threads, the execution time is compressed, significantly reducing wall-clock latency. This distinction is essential: while classical Big O complexity provides upper bounds on computational steps, the improvements described here concern practical runtime optimization which is critical for distributed, latency-sensitive systems such as the one targeted by this work. As discussed in [27], the performance of large-scale distributed systems is dominated by the behavior of the slowest components ("tail latency"), and variability becomes a critical challenge. Consequently, practical system optimization often targets improvements in response time and variability, rather than changes in classical algorithmic complexity.

4.4 Discussion and Limitations

This work set out to investigate performance limitations and architectural weaknesses in a system responsible for delivering targeted notifications, particularly in contexts involving dynamic group-based user resolution and real-time dispatch guarantees. Through detailed empirical analysis and focused experimentation, a clear understanding of system behavior under stress was established, leading to a set of targeted improvements across several technical dimensions.

A central theme was the identification and mitigation of dominant latency sources. The group resolution process, particularly the interaction with the CERN Authorization Service, emerged as the primary contributor to poor responsiveness. However, rather than addressing the symptom in isolation, the analysis exposed a broader picture of interdependent inefficiencies, including sequential execution paths, redundant data fetches, and suboptimal data structures.

To this end, multithreading was introduced to decouple slow I/O-bound operations and better utilize available system resources. While group expansion was the most impactful use case for this change, the broader introduction of concurrent patterns laid the foundation for a more scalable and reactive architecture. This was complemented by data structure optimization, specifically replacing linear search constructs with sets and set-based operations. These changes improved clarity, reduced cognitive overhead, and significantly lowered runtime cost for filtering logic.

Caching was examined as a general performance strategy, particularly effective for expensive or repetitive operations such as group membership lookups. Although not fully integrated into the production path, the feasibility study and preliminary designs showed promising potential in reducing the external dependencies and improving throughput. Importantly, the discussion recognized the trade-offs introduced by caching, including data freshness and invalidation complexity.

Another notable contribution was the role of observability and tracing. Fine-grained instrumentation enabled precise bottleneck localization and guided the iterative refinement process. This supports the broader claim that systematic observability is not merely a diagnostic tool but a critical design asset in evolving complex systems.

The system enhancements proposed and prototyped in this work, ranging from multithreaded processing to improved data semantics and concurrency safety, were designed to improve performance and maintainability. While each change addressed a specific concern, their combined effect is expected to result in a more robust, efficient, and transparent architecture.

The findings emphasize that performance optimization is rarely about isolated fixes. It is an investigative process that requires a comprehensive system understanding, rigorous validation, and careful balancing of correctness, performance, and complexity.

The development of this work faced limitations that are worth acknowledging. These constraints have implications for the system's performance, scalability, and practical application for CERN's infrastructure.

External Service Dependencies

The system's performance remains tightly coupled to the latency and availability of the CERN Authorization Service. Despite efforts to mitigate this dependency through caching mechanisms and parallelization techniques, the system remains vulnerable to potential outages or slowdowns in this external API. Such events could significantly degrade routing performance and overall system responsiveness.

Python's Global Interpreter Lock

The implementation's reliance on Python introduces limitations related to the language's Global Interpreter Lock (GIL) [28]. While the thread-based parallelism employed in the system effectively improves performance for I/O bound operations such as HTTP requests, the GIL restricts true concurrent execution for CPU-bound tasks. This design constraint may prevent the system from fully exploiting multi-core architectures when handling computationally intensive workloads, potentially limiting scalability under high computational demands.

Trade-offs in Caching

The proposed caching strategy for group memberships operates under the assumption that group compositions remain relatively stable over time. In scenarios where membership changes occur frequently, such as in dynamic team environments, stale cache entries could lead to incorrect targeting unless aggressively invalidated. This presents a trade-off between performance benefits from caching and the potential for outdated authorization decisions.

Furthermore, due to time constraints, cache size and eviction policies were not empirically tuned for production-scale workloads. Optimal cache configuration would require extensive testing under realistic conditions to determine ideal parameters for maximum efficiency without excessive resource consumption.

Implementation and Evaluation Constraints

Several proposed prototype features could not be fully implemented and evaluated within the system due to a combination of factors. Time restrictions played a significant role, but equally important was limited access to CERN resources. This constraint prevented comprehensive testing and benchmarking of the system under realistic conditions.

Testing certain proposals, particularly those involving parallel CERN Authorization Service requests, presented additional challenges. As this is a critical production system at CERN, there are valid concerns about the potential impact of overworking it with test requests. Such testing could potentially affect

the performance and stability of a key production system, making empirical evaluation difficult without dedicated testing environments.

Performance Monitoring Considerations

It is worth noting that there could be some performance overhead introduced by tracing mechanisms, which were not comprehensively analyzed in this study. While essential for monitoring and debugging, code instrumentation and tracing infrastructure may add computational and network overhead that should be carefully evaluated if the system is adopted into the real production environment. Future work should include a detailed assessment of the possible performance impact of tracing components and potential optimizations to minimize it.

Summary

This chapter demonstrates how thoughtful, multi-layered optimizations, grounded in a precise understanding of the system's internal data flow, concurrency potential, and dependencies on external services, can lead to meaningful and informed solution design for performance and scalability. Through detailed empirical analysis of carefully constructed test cases, performance bottlenecks were systematically identified, most notably the latency introduced by group membership resolution via the CERN Authorization Service. The findings informed a series of optimization strategies, including data structure replacements for faster lookups, introduction of parallelism where I/O latency dominated, immutability enforcement for thread safety, and theoretical evaluation of caching mechanisms. The cumulative effect of these targeted interventions illustrates the importance of a holistic and data-driven approach to performance engineering in distributed, latency-sensitive systems.

5

Conclusion and Future Work

Contents

This work undertook a systematic optimization of the CERN Notifications System, targeting performance bottlenecks in its routing components, a critical subsystem responsible for scalable, low-latency message delivery. Through empirical trace analysis, targeted code improvements, and architectural refinements, it has been made clear that the project stands to gain in responsiveness while preserving system correctness and maintainability.

Key outcomes demonstrate that:

Trace-driven optimization is indispensable for modern distributed systems. By instrumenting the router, this work identified hidden inefficiencies, such as sequential group resolution and list-based code flow, that might have eluded conventional profiling. The approach validates that observability is not merely diagnostic but foundational to performance engineering.

Practical optimizations need not require architectural overhauls. Strategic changes, replacing lists, introducing thread pools, and caching external calls, can yield significant latency reductions without redesigning the event-driven pipeline.

Real-world constraints shape technical trade-offs. While Python's GIL and external service dependencies imposed hard limits, the solutions adopted maximized gains within these boundaries.

Ultimately, this project exemplifies how incremental, data-driven optimization can elevate the performance of critical infrastructure. By reducing routing latency and improving scalability, the CERN Notifications System is now better equipped to tackle latency and meet any scalability demands, from routine alerts to time-sensitive emergency broadcasts. For CERN's staff, these improvements translate to more responsive communications, reduced notification delays during critical events, and enhanced reliability of the information pipeline that supports their operational activities.

5.1 Summary of Contributions

This work undertook a comprehensive investigation into the performance limitations of a production-grade message delivery system used in high availability environment. Through methodical profiling, code instrumentation, and benchmark-driven experimentation, the project systematically identified critical inefficiencies, particularly in user resolution logic and external service dependencies. Informed by these findings, targeted redesigns and prototype implementations were proposed and evaluated, addressing both algorithmic inefficiencies and architectural constraints. The key contributions of this work are outlined below:

Empirical Analysis

A series of controlled benchmark test cases was designed and executed to capture the system's runtime behavior under both realistic and stress-induced conditions. This analysis provided a holistic understanding of the system's internal workflows, performance characteristics, and dependencies. It revealed inefficiencies in user filtering, message dispatch, and most notably, group resolution and expansion. A broader value of this empirical study lies in how it can help guide the prioritization of redesign efforts and helped put forth prototype solution proposals across the system.

Data Structure Optimization

The use of list-based membership testing was a recurring inefficiency in the original. These were replaced by set operations to exploit constant-time lookup performance. This change drastically reduced execution time for filtering tasks, especially in sections involving set algebra. The adoption of set-specific methods also allowed for more concise and readable code.

Multithreading

Thread-level parallelism was introduced into several key workflows to improve responsiveness and resource utilization. The system was found to process I/O-bound operations sequentially, resulting in underutilized compute capacity and inflated latency. By employing `ThreadPoolExecutor` for concurrent execution, operations such as user data fetching, message dispatch, and group resolution were parallelized to reduce blocking time and improve throughput.

Thread-Safe Concurrency

In multithreaded components, mutable data structures posed a risk of race conditions and inconsistent reads. The introduction of `frozenset` provided a lightweight and semantically clear guarantee of immutability, making shared data safe for concurrent access without locking mechanisms.

Caching Strategy

The study explored caching as a performance enhancement technique, especially relevant in systems constrained by repeated access to external or computationally expensive resources. One key use case identified was group membership resolution, where repeated calls to an external service introduced avoidable delays. This strategy would preserve correctness while significantly reducing redundant network requests.

Prototype Redesign Proposals

Multiple prototype solutions were proposed targeting different performance bottlenecks. These included redesigned flows for computing user targets and dispatching messages, each emphasizing concurrency, immutability, and structural clarity.

Observability and Tracing Integration

Fine-grained runtime tracing was introduced to identify slow paths and API-level delays. This observability infrastructure proved critical in validating performance issues, especially those stemming from external service dependencies. The work supports the broader claim that observability is not merely a debugging tool but a foundational component for operating performant distributed systems.

5.2 Future Work

This work has focused on identifying and mitigating major performance bottlenecks in the CERN Notifications System, particularly within the routing component. However, several additional avenues remain to further evolve the system's responsiveness, scalability, and maintainability.

Phased Adoption

Importantly, any proposed enhancement should be adopted incrementally, with observability infrastructure being the first step. By putting in place detailed and reliable tracing across system components, beyond the routing layer, it becomes possible to quantify the impact of each change. Extending tracing coverage to the backend and consumer components would help build a comprehensive latency landscape, facilitating better architectural decisions and development prioritization.

Caching considerations

Caching has been put forward as a method to deliver meaningful gains, particularly for group resolution. However, deploying caching into production introduces new design questions, namely: what is the optimal eviction policy, what time-to-live should be applied, and how large the cache should grow. These parameters should be adjusted based on live traffic patterns and usage characteristics. To do this effectively, observability metrics such as cache hit rate, memory consumption, and eviction frequency should be collected and analyzed. Over time, this would allow tuning the cache for maximum performance with minimal overhead.

Threading considerations

Similarly, thread pool sizing used in various parallelized routines (e.g., user processing or page fetching for group members) should not be fixed arbitrarily. The number of threads in a `ThreadPoolExecutor` impacts both throughput and system resource usage, and may need to be tuned. Too few threads underutilize the system, while too many can cause context-switching overhead or API rate limiting. Empirical testing under realistic loads, combined with runtime instrumentation, will be essential to identify an optimal thread pool size.

Language

There are also runtime and language factors to consider. The router component currently runs on Python 3.6, a version that has reached end-of-life and no longer receives updates or security fixes [29]. Upgrading to a modern Python version (e.g., 3.10+) would unlock a variety of performance improvements, language features, and access to recent library versions. This upgrade would also improve compatibility with tools such as OpenTelemetry and improve maintainability going forward. In the longer term, with a more refined understanding of the routing component's complexity and requirements, it may be worthwhile to reimplement it in a compiled language such as Go. The Global Interpreter Lock (GIL) in Python limits concurrency in CPU-bound contexts, and Go's goroutine-based model provides lightweight concurrency primitives better suited for high-throughput, fan-out tasks like notification routing.

Queue mechanisms

Another area of interest is the behavior of the system under high load, particularly during critical events requiring rapid dissemination. In such cases, introducing queue purging or prioritization mechanisms could help ensure high-priority notifications bypass backlogged queues. This must be approached with caution since purging queues risks message loss for non-critical traffic, and should only be applied to specific types of urgent notifications.

Workload shift

It could also be beneficial to rethink the routing workload distribution. This would mean a more complex and nontrivial change to the system's architecture that would involve shifting some of the routing workload upstream, specifically, into the backend. Currently, the router is responsible for expanding channel memberships and group intersections at the moment of notification dispatch. However, channel membership changes are infrequent, while notification sending occurs frequently. Therefore, the backend could precompute and store expanded membership lists when a channel is created or updated. Simi-

larly, when a notification is sent targeting a known intersection (e.g., channel members ∩ group), and that combination has been previously resolved, the backend can reuse the cached expansion. This model introduces computational amortization, offloading frequent compute-heavy tasks from the runtime path and replacing them with low-latency database fetches by the router. If combined with a rewritten, efficient routing component (as discussed earlier), this dual optimization could yield substantial throughput improvements.

However, this strategy introduces new complexity: group membership at CERN is dynamic, and stored expansions may become stale. This would require background synchronization processes or periodic validation jobs to keep stored expansions in sync with the CERN Authorization Service, the source of truth.

Bibliography

[1] CERN Communications. (2019, Mar.) Migrating to open source technologies. European Organization for Nuclear Research (CERN). CERN News article. [Online]. Available: https://home.cern/news/news/computing/migrating-open-source-technologies

[2] R. A. Board, "Rss 2.0 specification," RSS Advisory Board, Tech. Rep., 2009, official RSS 2.0 Specification. [Online]. Available: http://www.rssboard.org/rss-specification

[3] R. Otto, "CERN Alerter - RSS based system for information broadcast to all CERN offices," CERN, Geneva, Tech. Rep., 2008. [Online]. Available: https://cds.cern.ch/record/1054455

[4] Antunes, Carina, Semedo, Jose, Wagner, Andreas, Ormancey, Emmanuel, Carpente, Caetan, and Jakovljevic, Igor, "Building a user-oriented notification system at cern," *EPJ Web of Conf.*, vol. 295, p. 05002, 2024. [Online]. Available: https://doi.org/10.1051/epjconf/202429505002

[5] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, p. 114–131, Jun. 2003. [Online]. Available: https://doi.org/10.1145/857076.857078

[6] R. Baldoni, R. Beraldi, S. Tucci Piergiovanni, and A. Virgillito, "On the modelling of publish/subscribe communication systems: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 17, no. 12, p. 1471–1495, Oct. 2005.

[7] Z. Hmedeh, H. Kourdounakis, V. Christophides, C. du Mouza, M. Scholl, and N. Travers, "Content-based publish/subscribe system for web syndication," *Journal of Computer Science and Technology*, vol. 31, no. 2, pp. 359–380, Mar. 2016. [Online]. Available: https://doi.org/10.1007/s11390-016-1632-8

[8] W3C, "WebSub: W3c recommendation for publish-subscribe on the web," World Wide Web Consortium, W3C Recommendation, 2018, version: 2018-01-23. [Online]. Available: https://www.w3.org/TR/websub/

[9] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, “Dapper, a large-scale distributed systems tracing infrastructure,” Google Research, Tech. Rep., 2010. [Online]. Available: https://static.googleusercontent.com/media/research.google.com/en//archive/papers/dapper-2010-1.pdf

[10] OpenTelemetry Community. Opentelemetry adopters. Cloud Native Computing Foundation. Hosted by OpenTelemetry. [Online]. Available: https://opentelemetry.io/ecosystem/adopters/

[11] OpenTelemetry Authors. (2023) Opentelemetry overview. Cloud Native Computing Foundation. [Online]. Available: https://opentelemetry.io/docs/specs/otel/overview/

[12] A. Janes, X. Li, and V. Lenarduzzi, “Open tracing tools: Overview and critical comparison,” 2023. [Online]. Available: https://arxiv.org/abs/2207.06875

[13] Jaeger Authors, “Jaeger documentation (v2.4),” https://www.jaegertracing.io/docs/2.4/, 2024, accessed: 2025-05-02.

[14] Zipkin Contributors, “Zipkin: Architecture overview,” https://zipkin.io/pages/architecture, 2024, accessed: 2025-05-02.

[15] SigNoz Contributors, “Signoz documentation: Introduction,” https://signoz.io/docs/introduction/, 2024, accessed: 2025-05-02.

[16] M. Nešić, “Evaluating the integration of distributed tracing signals into the cern monitoring infrastructure. cern openlab summer student lightning talks (2/2),” https://cds.cern.ch/record/2868468, 2023, presentation.

[17] C. Majors, L. Fong-Jones, and G. Miranda, *Observability Engineering*. O’Reilly Media, 2022. [Online]. Available: https://books.google.pt/books?id=KGZuEAAAQBAJ

[18] CNCF TAG Observability, “Tag observability whitepaper,” https://github.com/cncf/tag-observability/blob/main/whitepaper.md, 2022, accessed: 2025-05-02.

[19] K. Shahedi, H. Li, M. Lamothe, and F. Khomh, “Tracing optimization for performance modeling and regression detection,” 2024. [Online]. Available: https://arxiv.org/abs/2411.17548

[20] N. Ezzati-Jivan, Q. Fournier, M. R. Dagenais, and A. Hamou-Lhadj, “Depgraph: Localizing performance bottlenecks in multi-core applications using waiting dependency graphs and software tracing,” in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, Sep. 2020, p. 149–159. [Online]. Available: http://dx.doi.org/10.1109/SCAM51674.2020.00022

[21] I. Kohyarnejadfard, D. Aloise, S. V. Azhari, and M. R. Dagenais, “Anomaly detection in microservice environments using distributed tracing data analysis and NLP,” *Journal of Cloud Computing*, vol. 11, no. 1, p. 25, Aug. 2022. [Online]. Available: https://doi.org/10.1186/s13677-022-00296-4

[22] B. Eaton, J. Stewart, J. Tedesco, and N. C. Tas, “Distributed latency profiling through critical path tracing: Cpt can provide actionable and precise latency analysis.” *Queue*, vol. 20, no. 1, p. 40–79, Mar. 2022. [Online]. Available: https://doi.org/10.1145/3526967

[23] Z. Zvara, P. G. Szabó, B. Balázs, and A. Benczúr, “Optimizing distributed data stream processing by tracing,” *Future Generation Computer Systems*, vol. 90, pp. 578–591, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17325141

[24] CERN IT Department. (2023) Cern notifications service. European Organization for Nuclear Research (CERN). Official documentation and user portal for CERN’s notification system. [Online]. Available: https://notifications.web.cern.ch/

[25] Node.js contributors. (2023) The node.js event loop, timers, and process.nexttick(). OpenJS Foundation. Node.js v21 Documentation. [Online]. Available: https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick

[26] Jaeger Authors, *Jaeger: Getting Started*, Cloud Native Computing Foundation, Jun. 2021, documentation for Jaeger v1.24. [Online]. Available: https://www.jaegertracing.io/docs/1.24/getting-started/

[27] J. Dean and L. A. Barroso, “The tail at scale,” *Commun. ACM*, vol. 56, no. 2, p. 74–80, Feb. 2013. [Online]. Available: https://doi.org/10.1145/2408776.2408794

[28] Python Software Foundation, “Thread state and the global interpreter lock,” Python 3 Documentation, accessed: [Insert access date]. [Online]. Available: https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock

[29] P. S. Foundation, “Python developer’s guide: Python versions,” 2024, accessed: 2025-05-03. [Online]. Available: https://devguide.python.org/versions/