José Semedo
jose.francisco.semedo@gmail.com

Instituto Superior Técnico, Lisboa, Portugal

May 2025

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 Additionally, an outdated external API integration was modernized and parallelized to further reduce latency during group resolution.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

1. Introduction

The CERN Notifications System originated within the MALT project [3], an initiative aimed at replacing proprietary software dependencies with open, maintainable, internally developed alternatives. Within this strategic push for digital sovereignty, the need emerged for a unified, programmable messaging platform to coordinate communication across CERN's sprawling and heterogeneous community.

Initially a component-level service, CERN Notifications evolved into a critical standalone system designed to facilitate structured, scalable, and channel-agnostic dissemination of information. It enables services and user groups to programmatically target individuals or dynamically defined cohorts using multiple delivery mechanisms such as email, SMS, push, and chat services. The platform supports user configurability for delivery preferences and integrates with CERN's identity infrastructure, offering a high degree of control, flexibility, and interoperability.

With widespread adoption across operational and user-facing domains, CERN Notifications [1] has become the de facto communication backbone for scenarios ranging from user support to infrastructure alerts. However, as reliance on the service grows, so do the demands on its performance, particularly in high-fan-out, time-sensitive use cases.

This work addresses the need for performance improvement by focusing on the routing component of the system, the part responsible for resolving recipients given a targeting rule or group specification. Routing is a latency-critical phase in the message delivery pipeline, especially when resolving large or complex user groups. Through a data-driven performance analysis effort, including distributed tracing instrumentation and benchmark-based executing profiling, this work identifies core inefficiencies and proposes targeted optimizations. These include caching strategies, data structure revisions, and parallelization of sequential operations.

The overarching goal is not architectural overhaul, but systematic enhancement of the router's behavior under load, guided by empirical evidence. By doing so, this work contributes to ensuring the long-term scalability and responsiveness of CERN Notifications, maintaining its ability to support communication at the speed and scale demanded by a modern scientific institution.

The remainder of this document is structured as follows: Section 2 reviews related work on notifica-

tion delivery architectures and performance analysis techniques such as tracing and profiling. Section 3 outlines the current CERN Notifications architecture, emphasizing its components, data flow, and performance-relevant design. Section 4 details the experimental methodology, profiling results, and proposed optimizations targeting the routing component. Finally, Section 5 concludes with a summary of contributions and directions for future improvement.

2. Related Work

The design and optimization of modern notification systems draws upon decades of research and practical implementations in information distribution architectures. This section examines the evolution of these systems, from early web syndication protocols to contemporary event-driven messaging frameworks, and discusses the role of observability tools in performance optimization.

RSS (Really Simple Syndication) was one of the earliest technologies adopted for automated web content distribution, offering a decentralized and minimal-infrastructure approach to publishing updates. At CERN, it gained traction due to its open standards, cross-platform client support, and ease of deployment, making it a suitable choice for initial efforts in disseminating institutional announcements and alerts. The CERN Alerter system was built around this model, using structured polling to detect and notify users of new information.

Despite its early utility, RSS revealed several critical limitations as CERN's communication needs evolved. Its fundamental pull-based mechanism required clients to periodically poll servers for updates, introducing inherent delays in message delivery and consuming resources unnecessarily, even when no new content was available. This made the system inefficient and unsuitable for time-sensitive use cases, such as operational alerts or emergency notifications, where real-time responsiveness is essential.

Furthermore, RSS lacked support for user-specific targeting, prioritization, or delivery customization. All subscribers received the same content regardless of relevance, urgency, or device constraints. In an organization as diverse and role-driven as CERN, this broadcast model became increasingly inadequate. The stateless nature of RSS, combined with its limited integration capabilities, particularly as computing platforms diversified beyond Windows, further hindered its scalability and maintainability.

These constraints highlighted the need for a more modern and flexible communication infrastructure. The shift from RSS-based polling to push-oriented, event-driven architectures reflects this evolution, enabling systems that are more responsive, resource-efficient, and capable of delivering personalized and context-aware notifications at scale.

Publish-Subscribe Systems emerged as a more scalable and flexible alternative for distributing information [6] as dynamic environments like CERN outgrew the limitations of polling-based approaches. 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 [2]. 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 have a prior agreement on the topic structure, and cannot perform dynamic, context-aware filtering of content.

To address this, content-based pub/sub systems were proposed [8]. 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.

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.

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 [18]. 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 sends 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 suitable for the complex, institution-wide notification requirements at CERN. The CERN Notifications system does more than simply push content, it routes messages based on a rich set of user-defined rules, delivery contexts, group memberships, and dynamic filters. WebSub lacks native support for such intermediate decision-making, blindly forwarding updates from publishers to all subscribers without regard for relevance, delivery conditions, or user context. Furthermore, its reliance on HTTP webhooks as the sole delivery mechanism limits its applicability in environments like CERN, where notifications must be delivered via diverse channels such as SMS, email, push services, and internal messaging platforms, each with distinct protocols, failure modes, and delivery guarantees. WebSub's design offers no means to accommodate this heterogeneity or support critical delivery features like prioritization or preference-aware routing.

System Profiling and Performance 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 [17] 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 [15]. OpenTelemetry [14] provides vendor-agnostic APIs and SDKs for capturing metrics, logs, and traces, enabling comprehensive insight into system performance with minimal vendor lock-in.

Attention is focused on 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 [9] reviews over 30 such tools and highlights the diversity in tracing capabilities and implementations.

Jaeger , originally developed at Uber and now maintained by the CNCF, is a production-grade distributed tracing platform designed for high-scale microservices and event-driven systems. It supports trace ingestion, storage, querying, and visualization, with features like service dependency graphs and latency analysis. Jaeger's modular architecture supports various backends and horizontal scaling. Its native compatibility with OpenTelemetry ensures low integration overhead and future-proof observability. These qualities, combined with its maturity and alignment with CERN's existing CNCF-based infrastructure, made it a natural choice for deployment.

Zipkin inspired by Google's Dapper and developed by Twitter, is a lightweight distributed tracing system known for its simplicity and low overhead. While effective for basic latency tracing, it lacks advanced features like high-cardinality tagging, dynamic sampling, and full OpenTelemetry support. Its limited scalability and slower development pace make it less suitable for complex, high-scale environments like CERN, though it remains useful in lightweight or development settings.

SigNoz is a modern, open-source observability platform built around OpenTelemetry, offering integrated support for logs, metrics, and traces with a developer-friendly UI. It aims to be a full-stack alternative to tools like Datadog, enabling unified observability through a single dashboard. While promising and feature-rich, its relative immaturity and smaller community make it less suitable for CERN's production-grade requirements at this stage.

Among the evaluated solutions, Jaeger emerged as a strong candidate for integration within CERN's monitoring infrastructure [12]. Its open-source foundation, scalability, and native compatibility with OpenTelemetry make it particularly well-suited to the demands of large-scale, distributed environments. The combination of OpenTelemetry and Jaeger aligns with CERN's engineering principles, favoring vendor-neutral, ecosystem-compatible technologies, which simplifies integration and promotes maintainability. This strategy leverages existing familiarity and infrastructure support, reducing operational overhead and increasing adoption potential

across projects.

The integration of tracing as a core observability mechanism further enhances the system's maintainability, debuggability, and scalability. Observability, as emphasized by Majors et al. and the CNCF, is not merely about monitoring but about enabling engineers to infer internal system states from external outputs [11, 4]. Recent research underscores the value of tracing in operational intelligence. Shahedi et al. [16] show how statistical models applied to trace data can isolate performance regressions without processing entire trace sets. Similarly, Ezzati-Jivan et al. [7] use dependency graphs from software traces to identify concurrency bottlenecks, and Ibidunmoye et al. [10] apply NLP techniques to trace logs for anomaly detection. Alizadeh et al. [5] describe critical path tracing as a method for revealing latency-dominant execution paths, while Ostermann et al. [19] show that performance tuning guided by trace data can yield significant gains in stream processing systems. Together, these studies validate the central role of tracing in optimizing and understanding complex, distributed workflows.

3. Architecture

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 1 is an architecture overview.

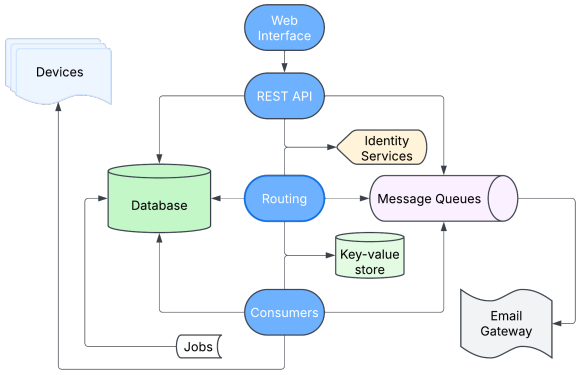

Figure 1: Existing Architecture

Web Interface is a web-based portal that serves as the primary interface for interacting with the system. Through it, users can configure channels, set preferences, register devices, and issue notifications. This interface communicates with the backend over a REST API, providing a clean separation between frontend interactions and server-side logic.

Backend Server exposes a RESTful API consumed by both users and system components. It performs authentication and authorization checks, validates incoming requests, and coordinates operations such as modifying channel configurations, updating preferences, or initiating notifications. Upon receiving a valid notification request, it executes the necessary processing, logs relevant audit information, and publishes the message to the routing queue.

Router handles the most computation-heavy logic in the pipeline. It processes messages from the queue and applies multi-layered filtering and expansion logic. This includes resolving group memberships, applying user-defined preferences, honoring mutes, and generating individualized recipient lists. A single incoming notification can result in thousands of customized messages, each routed to its appropriate delivery queue for downstream handling.

Consumers components exist for each delivery mechanism, such as email, SMS, push, or chat platforms. These consumers handle their queues, generate the appropriate payloads, and deliver messages to the final endpoints using channel-specific protocols and error-handling routines.

Data Persistence is handled by a PostgreSQL database, hosted by CERN's infrastructure, to persist all key data: user and channel definitions, preferences, mute states, notification records, and supporting metadata. Additionally, an etcd-based key-value store is employed to support operation deduplication and to maintain an auditable log of state transitions, particularly valuable in failure recovery scenarios, and security reviews.

Identity Integration which encompasses user identity and group memberships, are managed centrally by CERN's identity service. It provides authoritative data on accounts and access rights. This external service is critical for enforcing access control policies and resolving group-channel associations. Backend and routing components interact with it via a dedicated API to determine which users belong to which groups and, consequently, who should receive specific notifications.

Architecture considerations

Although the system's architecture emphasizes modularity and scalability, each component introduces its own set of performance considerations. The backend server, implemented in Node.js [13], leverages its asynchronous, event-driven architecture to efficiently manage high volumes of concurrent requests. Thanks to the non-blocking I/O model, operations such as database interactions or message queue insertions are handled asynchronously and with minimal overhead, ensuring that the backend remains responsive even under heavy load.

The consumer services, which are tasked with delivering notifications to their respective endpoints, are built to scale horizontally. Each message retrieved from their queues is processed independently, allowing multiple instances of a consumer to run in parallel without risk of conflict or shared state issues. This design is particularly effective during peak activity periods, as increasing the number of consumer instances leads directly to higher processing throughput, improving responsiveness without requiring changes to system complexity.

The router component, however, poses more significant performance challenges. While it too, operates on a message queue, its task is considerably more compute-intensive. Routing involves resolving dynamic group memberships, evaluating complex filtering rules, and performing external lookups, all of which can result in the fan-out of a single message to thousands of individualized deliveries. These operations are stateful, interdependent, and sensitive to input scale and structure. In situations where messages target large or deeply nested groups, effectively triggering mass delivery across the user base, the router becomes the system's limiting factor. Routing logic requires specialized performance tuning and analysis, the details of which are discussed in the subsequent chapters.

To gain clearer insight into the system's internal behavior and the expansion process it performs, consider the lifecycle of a notification directed at a group, as illustrated in Figure 2. The process begins when the backend server receives a request to dispatch a notification. This request is first validated, authorization is verified, and an audit entry is recorded. Any necessary updates to the database are applied, after which a single message representing the notification is placed onto the router message queue.

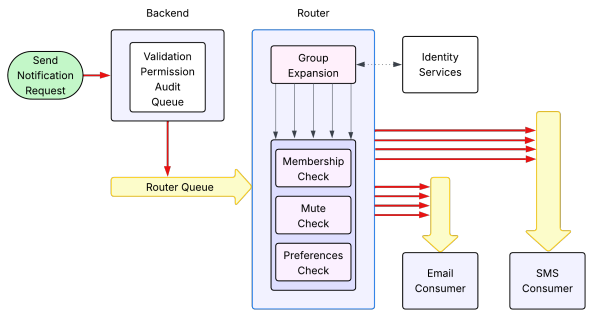

Figure 2: Notification Flow

Once the router retrieves this message, it initiates the expansion phase by resolving the target group into its members. This step involves querying the external identity service to retrieve up-to-date group membership data and obtain the relevant user accounts. For each resolved user, the system evaluates their delivery preferences and registered devices. Depending on user configuration, multiple delivery channels—such as email, SMS, or other supported platforms—may be applicable. Consequently, several discrete message objects may be generated per user, each representing a unique delivery route.

These individualized messages are then distributed into the appropriate message queues, segmented by delivery type. Each consumer component subsequently retrieves and processes its corresponding messages, executing delivery via its designated medium. This end-to-end workflow, starting from a single group-targeted notification, results in a significant amplification of delivery tasks, underlining the workload implications introduced by group expansion and per-user customization.

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.

4. Assessment

4.1. Setup and Methodology

To ensure a stable and isolated environment for testing, the system is deployed on a dedicated Ubuntu machine. This minimizes interference from unrelated background processes and establishes a reproducible foundation for performance measurements. The router component, central to the performance evaluation, is deployed paired with a local Jaeger all-in-one instance, which provides distributed tracing capabilities. Jaeger enables the collection of fine-grained trace data, including span-level timing information, offering detailed visibility into the internal behavior of the router.

To replicate a realistic operating environment and preserve functional parity with production, essential supporting services such as the PostgreSQL database and the etcd key-value store are also hosted locally. This setup recreates the key architectural dependencies relevant to the router's behavior, providing an accurate testbed for evaluating its performance characteristics.

The testing approach centers on a series of controlled workload scenarios designed to stress different aspects of the router's logic. Tests are triggered either through the user-facing interface or by directly submitting API requests to the backend, both resulting in the enqueuing of a routing message. Test parameters vary across key dimensions, including the number of users and/or groups in a channel and the characteristics used for filtering or matching recipients. These variables are selected to expose performance limits under realistic and varied conditions. Each scenario corresponds to a distinct combination of these parameters. An overview of the test scenarios is presented in Table 1.

4.2. Timing Results and Performance

To provide context for the performance of the current system implementation, this subsection reports the

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 1: Test cases

total execution time recorded for each of the defined test scenarios. The measurement reflects the time taken by the router from the moment a notification is received until all intended recipients have been fully processed for message delivery. The collected results are summarized in Table 2.

Table 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 execution times across test cases reveal key performance drivers. Test cases 1 and 2, based on direct user and group memberships, show low latency, with test 2 taking slightly longer due to more entities, indicating that cost scales with the number of distinct members.

Test case 3, which introduces intersection targeting with a large group, results in a sharp performance drop, suggesting that intersection logic over a large user base is costly. Test case 4, using the same logic but a smaller group, runs much faster, reinforcing that group size is an important factor, not only the intersection mechanism itself.

Test cases 5 and 6 show the highest execution times, both involving large group targets. Despite differing channel structures, one with a member-group, the other with many member-users, their similar runtimes suggest that total group size, rather than composition, drives performance impact.

Test case 1 - multiple users

Trace analysis of this test reveals a clear candidate for a performance bottleneck: the group fetching and expansion phase, which dominates the total execution time in Figure 3.

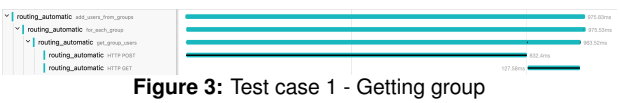

Figure 3: Test case 1 - Getting group

Test case 2 - multiple users and groups

This test introduces multiple groups, causing the time to grow linearly with the number of groups. Figure 4 illustrates the cumulative time caused by the group resolution requests.

Figure 4: Test case 2 - Getting groups

In this test, another candidate bottleneck issue is shown more clearly. The iteration of each of the members that it has resolved to in the previous step adds up to a non-insignificant amount of time. This can be observed in Figure 5.

Figure 5: Test case 2 - Iterating Users

Test case 3 - multiple users intersect large group

This test case introduces intersection targeting and further confirms that resolving group members via CERN's authorization service remains the primary performance bottleneck. As shown in Figure 6, the system issues multiple consecutive HTTP GET requests to the authorization API, each ranging from 600 ms to over 1100 ms. These paginated responses cumulatively introduce several seconds of delay, particularly significant for large groups.

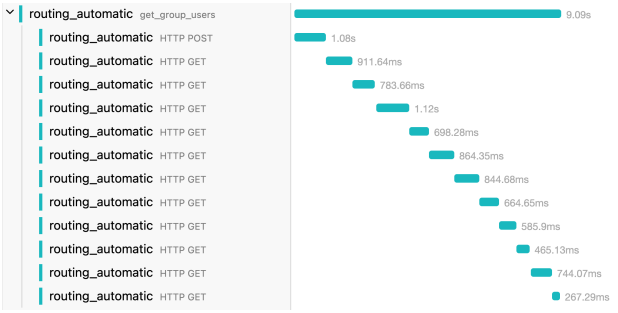

Figure 6: Test case 3 - Paginated Group Return

As in previous tests, once group resolution completes, the system proceeds with a series of user-level database lookups. While each individual query remains fast (1–3 ms), the increased number of users in this test amplifies the cumulative cost, resulting in a notable 34 seconds of additional execution time.

Test case 4 - multiple users intersect small group

Test case 4 reduces the size of the intersecting group compared to the previous test, resulting in a

noticeably shorter runtime. However, group resolution still accounts for the dominant portion of total execution time, 1.42 seconds overall, with the `add_users_from_groups` span alone consuming 947 ms (Figure 7). This reaffirms group membership expansion as a major latency source, regardless of intersection.

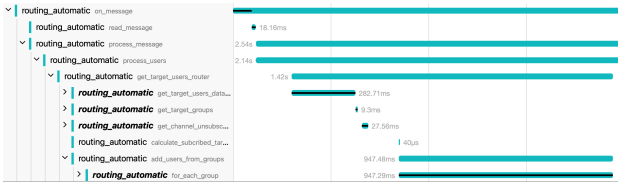

Figure 7: Test case 4 - Starting Segment

Test case 5 - large group intersect large group
Test Case 5 is highly informative, it is the first of the two high-latency benchmarks, designed to simulate a stress scenario where both the channel and the intersection target involve large user groups. This mirrors realistic high-volume notification cases, particularly when intersecting broad organizational units, common when attempting to reach as much of the user base as possible through overlapping groups.

The queried group's large size leads to multiple paginated responses from the CERN Authorization Service. Each HTTP request adds several hundred milliseconds of delay, and due to their strictly sequential execution, the cumulative latency reaches several seconds (Figure 8). This reaffirms group resolution as a fundamental bottleneck in the pipeline.

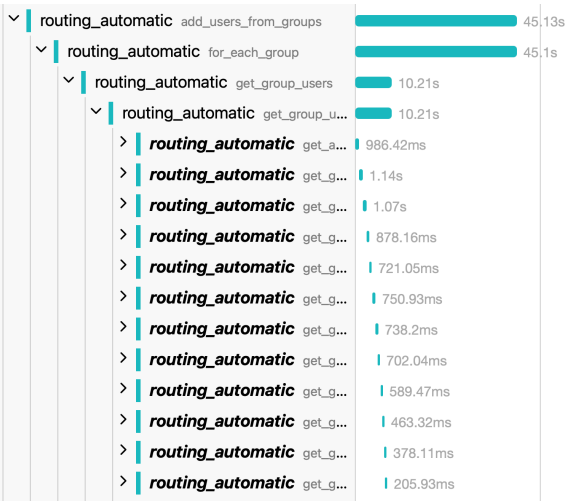

Figure 8: Test case 5 - Resolving Group Users

However, this is not the only issue. After resolving memberships, the system computes the intersection between two large and overlapping groups, producing a high-volume user set. This triggers a second performance-intensive phase: per-user processing. The trace illustrates a dense sequence of spans, one for each user, involving object preparation, preference, mute checks, and database lookups. Though

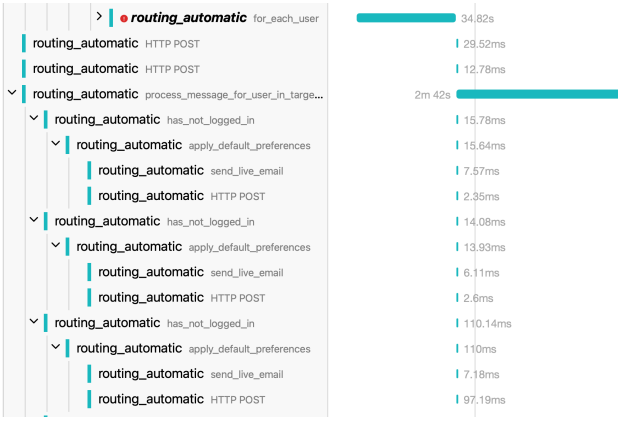

Figure 9: Test 5 - Per user processing

individually fast, these operations accumulate substantial overhead when executed thousands of times, significantly contributing to the overall runtime (Figure 9).

Test case 6 - multiple users intersect large group
This test case mirrors Test Case 5 in overall scale, involving a high number of users both in the channel and the target group. The key difference is structural: the large member group is replaced with a large set of direct member users, while still targeting a sizable group. Despite this change, total execution time remains comparable.

As before, group resolution via CERN's Authorization Service is the dominant performance bottleneck. The trace also shows the familiar sequential per-user processing pattern after resolution. Although the absence of a large group slightly reduces the load during the initial expansion phase, the large volume of targeted users and the cost of repeated per-user operations keep the total runtime similar.

4.3. Performance Analysis and Solution Proposals

Inefficient Sequential Processing
A major factor behind poor performance in large-scale tests (e.g., cases 3, 5, and 6) is the router's strictly linear execution model. This design forces all user and group-level operations, such as group expansion, user validation, preference checks, and mute evaluations, to execute serially, even when tasks are independent.

Listing 1: router.py - get_channel_subscribed_users

```
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
```

While individual operations are lightweight, their cumulative cost becomes significant when repeated thousands of times. This is especially evident in functions like `get_channel_subscribed_users` and `get_group_users_api` (Listings 1 and 2), where users and groups are processed one at a time despite being eligible for concurrent handling.

Listing 2: authorization.py - get_group_users_api

```
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
```

Group targeting logic further illustrates the inefficiency, as both group and user expansions occur sequentially with no parallel API calls. The lack of parallelism leads to poor scalability. As notification targets grow, routing time increases linearly, making the system unsuitable for high-volume or latency-sensitive environments.

Parallelization Proposal

To address this, critical sections of the routing logic were refactored as part of the prototype solution proposal, using Python's `ThreadPoolExecutor` (Listing 3) to execute independent tasks concurrently. This change was applied to operations such as group resolution (`add_users_from_group`) and per-user processing, where each task can safely run in parallel and benefits from overlapping I/O waits.

Listing 3: 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)
```

Conceptually, parallelization shifts the effective wall-clock complexity from $\mathcal{O}(N \cdot T)$ to approximately $\mathcal{O}(max(T))$, assuming ideal conditions and no bottlenecks. While this is not a true complexity reduction, it offers practical performance gains in latency-sensitive scenarios.

Membership Testing

A key inefficiency in current routing logic stems from the overreliance on Python's `list`. This is especially true for membership checks, where `set` would provide significantly better performance and scalability. This choice negatively impacts runtime efficiency, especially under high user volumes or in multithreaded scenarios.

Membership tests like `x in list` have linear time complexity $\mathcal{O}(N)$. When used repeatedly within loops or user resolution logic—as seen in `get_target_users` (Listing 4), the cost becomes quadratic with respect to the number of users. This adds substantial latency when lists grow into the hundreds or thousands of entries.

Listing 4: 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)
```

Additionally, lists are not thread-safe and require explicit locking for safe concurrent access, complicating any move toward parallelism. Sets, in contrast, support constant-time membership checks and

are better suited for concurrent workloads where frequent lookups dominate.

Leveraging Sets Proposal

Replacing lists with `set` or `frozenset` improves both efficiency and correctness. Sets offer average-case constant-time membership checks ($\mathcal{O}(1)$), a significant improvement over the linear-time lookups ($\mathcal{O}(N)$) required by lists. This becomes critical in areas where membership checks are frequent, such as filtering unsubscribed users or resolving group targets.

Routing data, e.g., user IDs, group names, is typically unique and hashable, making it well-suited for set-based operations. Sets also inherently prevent duplicates, simplifying logic and ensuring correctness when merging user collections from multiple groups.

In concurrency contexts, `frozenset` is preferred due to its immutability. It avoids mutation-related race conditions and can be safely passed across threads or reused in caching.

Set operations also align well with routing logic. Tasks like removing unsubscribed users from a target pool using `difference_update` are both faster and more expressive. As shown in Listing 5, this avoids redundant iteration and scales predictably with input size.

Listing 5: 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
```

Dominant Bottleneck - Group Resolution

As observed during testing (especially in test cases 3, 5, and 6), group membership resolution via the CERN Authorization is a major bottleneck. It is done through synchronous, paginated (for large groups) HTTP requests, processed sequentially and blocking per group. This results in significant cumulative latency.

As shown in Listing 1, each group is resolved one at a time, with each page fully fetched and processed before the next. No parallelism is used, and all API interactions block execution.

This design tightly couples routing performance to the responsiveness of an external service, creating a critical scalability and latency issue, especially in time-sensitive scenarios like alerting.

Caching and Concurrency Proposal

Full replication of the group database is infeasible, but caching provides a practical compromise. By memoizing previously fetched group memberships, the system can avoid redundant API calls. Python's `functools.lru_cache` can be used for this purpose, keyed by group ID. As shown in Listing 6, the entire paginated result is assembled and cached as a `frozenset`, ensuring immutability, hashability, and thread safety.

Listing 6: 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(
        prepare_user(member)
        for member in all_members
        if prepare_user(member) is not null
    )
```

The previous group resolution relied on the now-deprecated endpoint, which was replaced with a new fully supported endpoint. The updated API response includes detailed pagination metadata, such as total, offset, and limit, enabling clients to determine the total number of result pages and their boundaries. This metadata is critical for implementing more advanced retrieval strategies, such as non-sequential access or parallel fetching of paginated results.

With pagination metadata available from the updated endpoint, a solution proposal was designed to parallelize intra-group resolution. After fetching the first page and extracting pagination parameters, the remaining pages are calculated and retrieved concurrently using multiple threads or asynchronous calls. Each page is processed independently, and user objects are built in parallel using existing transformation logic. This approach compresses wall-clock latency from linear ($\mathcal{O}(P \cdot T)$) to a possibly near-constant ($\mathcal{O}(T)$), where P is the number of pages and T is the time to retrieve one. This enhancement is especially beneficial for large groups, where sequential page retrieval previously imposed a significant delay.

5. Conclusions

This work aimed to uncover and design solution proposals to address performance bottlenecks and architectural limitations in a system designed to deliver targeted notifications, especially in scenarios involving dynamic, group-based user resolution and the need for timely dispatch. Through empirical analysis and controlled experimentation, it achieved a detailed understanding of system behavior under load, leading to a set of focused prototype proposal enhancements across several technical layers.

One of the central findings was the identification of group resolution, particularly the interaction with the CERN Authorization Service, as a primary source of latency. However, the performance issues extended beyond this single factor, revealing a series of interrelated inefficiencies, including linear, sequential processing, redundant external lookups, and inefficient data structures. These were addressed not through isolated fixes, but by rethinking broader execution patterns and data handling mechanisms.

Multithreading was introduced as a mechanism to decouple slow I/O operations and take advantage of available compute resources. While the group expansion process can benefit most visibly from this change, the adoption of concurrent execution paradigms means a broader shift toward a more scalable and reactive system model. This was further reinforced by the replacement of inefficient list-based logic with set-based operations, simplifying filtering and estimated to reduce processing overhead, especially in high-volume scenarios.

Caching was also evaluated as a performance strategy, focusing on reducing the cost of repetitive operations such as group membership queries. The analysis carefully considered trade-offs related to cache freshness and invalidation, recognizing the operational complexity that such mechanisms may introduce.

The use of observability and distributed tracing played a critical role in guiding the optimization process. Detailed instrumentation within the routing component allowed for precise and informed bottleneck identification. This underscored the value of observability not just for debugging, but as a design principle that supports ongoing system evolution.

For CERN's Notification System, the adoption of these changes can translate to faster, more reliable message routing—crucial for time-sensitive communications ranging from routine updates to emergency alerts—ultimately enhancing the organization's operational responsiveness.

References

[1] 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.*, 295:05002, 2024.

[2] R. Baldoni, R. Beraldi, S. Tucci Piergiovanni, and A. Virgillito. On the modelling of publish/subscribe communication systems: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(12):1471–1495, Oct. 2005.

[3] CERN Communications. Migrating to open source technologies, Mar. 2019. CERN News article.

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

[5] 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*, 20(1):40–79, Mar. 2022.

[6] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[7] 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)*, page 149–159. IEEE, Sept. 2020.

[8] 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*, 31(2):359–380, Mar. 2016.

[9] A. Janes, X. Li, and V. Lenarduzzi. Open tracing tools: Overview and critical comparison, 2023.

[10] 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*, 11(1):25, Aug. 2022.

[11] C. Majors, L. Fong-Jones, and G. Miranda. *Observability Engineering*. O'Reilly Media, 2022.

[12] 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.

[13] Node.js contributors. The node.js event loop, timers, and process.nexttick(), 2023. Node.js v21 Documentation.

[14] OpenTelemetry Authors. Opentelemetry overview, 2023.

[15] OpenTelemetry Community. Opentelemetry adopters. Hosted by OpenTelemetry.

[16] K. Shahedi, H. Li, M. Lamothe, and F. Khomh. Tracing optimization for performance modeling and regression detection, 2024.

[17] 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. Technical report, Google Research, 2010.

[18] W3C. WebSub: W3c recommendation for publish-subscribe on the web. W3c recommendation, World Wide Web Consortium, 2018. Version: 2018-01-23.

[19] Z. Zvara, P. G. Szabó, B. Balázs, and A. Benczúr. Optimizing distributed data stream processing by tracing. *Future Generation Computer Systems*, 90:578–591, 2019.