A Performance Comparison of Authentication and Authorization Patterns for Microservices Applications

Rafael F. Cardoso, Jéferson C. Nobre

Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Porto Alegre – RS – Brasil

{rfcardoso, jcnobre}@inf.ufrgs.br

Abstract. *The microservices architecture has gained prominence in modern software development due to its flexibility, scalability, and resilience. However, ensuring robust security measures within microservices environments remains a challenge. This paper presents an empirical study evaluating authentication and authorization patterns for microservice-based applications. Three distinct versions of a baseline application were developed, each implementing a different authentication and authorization pattern: edge-level, centralized service-level, and decentralized service-level. Performance and resource consumption metrics were collected and analyzed across API endpoints. Results indicate that decentralized mechanisms generally outperform centralized approaches in terms of response time and efficiency, although they are associated with a higher storage cost.*

1. Introduction

The microservices architecture represents a paradigm shift in software design and development, epitomizing the principle of modularity. At its core, it advocates for the decomposition of complex applications into smaller, loosely coupled services, each encapsulating a specific functionality or business capability [Newman 2015]. Unlike traditional monolithic architectures, where entire applications are built as a single, cohesive unit, this architecture promotes the notion of independently deployable services, each with its own distinct purpose and responsibility.

The essence of microservices lies in their emphasis on autonomy, enabling development teams to work on individual services in isolation without being constrained by the dependencies inherent in monolithic architectures [Fowler 2014]. This autonomy fosters rapid iteration, allowing teams to deploy updates and enhancements to specific services without affecting the functionality of the entire application.

Microservice-based systems present unique complexities that require a nuanced approach to security. The primary challenge is the increased attack surface resulting from the proliferation of network endpoints and communication channels between services [Dragoni et al. 2017]. Each microservice exposes its own API, increasing the potential entry points for malicious actors to exploit vulnerabilities and launch attacks. The motivation for this study lies in addressing these security challenges, particularly focusing on the critical aspects of authentication and authorization.

Minimizing overhead and maximizing performance are critical when securing microservices. Given their dynamic nature and frequent operation in resource-constrained

environments [Fernando and Wickramaarachchi 2022] [Heinrich et al. 2017], optimizing security processes is essential.

While the significance of securing microservices is widely acknowledged, there remains a critical gap in understanding the performance trade-offs introduced by different authentication and authorization patterns in these environments. Security mechanisms that are overly centralized may introduce bottlenecks, negatively impacting the responsiveness and scalability of services. Conversely, highly decentralized approaches, while potentially reducing latency, may complicate management and introduce consistency risks. Despite the abundance of theoretical discussions and best practice recommendations, there is a lack of empirical data comparing the practical performance implications of these different patterns.

This study is designed with the overarching objective of empirically assessing and contrasting the performance implications across four versions of a representative microservices application, with each version deploying a different pattern for providing authentication and authorization. By evaluating the outcomes, we seek to determine which pattern best balances security requirements while minimizing performance overhead in microservices architectures.

2. Background

2.1. Existing Authentication and Authorization Patterns

Centralized authentication and authorization mechanisms involve consolidating authentication and authorization procedures within a dedicated service or identity provider [OWASP 2017]. Typical implementations include OAuth 2.0, OpenID Connect, and LDAP-based systems, where a central authority is responsible for issuing tokens or credentials that services validate before granting access. This pattern offers unified control and management of user identities and permissions, simplifying administration and enforcement of access control policies. However, centralized solutions may introduce single points of failure and scalability bottlenecks, particularly in large-scale distributed systems.

Decentralized mechanisms distribute authentication and authorization procedures across individual services, allowing each service to independently manage user identities and permissions [OWASP 2017]. Examples of this pattern include local JWT validation by services or services maintaining their own ACL (Access Control Lists) or RBAC (Role-Based Access Control) configurations internally. This pattern offers greater autonomy and flexibility, enabling services to adapt to changing requirements and scale independently. However, decentralized solutions may result in inconsistent enforcement of access control policies and increased complexity in managing distributed identities.

Edge mechanisms offload authentication and authorization procedures to the network edge, typically at the API gateway, service mesh ingress controller, or reverse proxy [OWASP 2017]. This pattern offers low latency and efficient access control enforcement at the perimeter, reducing the burden on individual services by ensuring that only authenticated and authorized requests reach them. However, edge solutions may introduce dependencies on network infrastructure and offer limited flexibility in enforcing fine-grained, context-aware access control policies within services themselves, which may still be necessary for certain sensitive operations.

2.2. Technological Landscape

Containerization technologies, such as Docker, have revolutionized the way distributed applications are packaged and deployed [Guerrero et al. 2018]. Containers provide lightweight, isolated environments that encapsulate individual services and their dependencies, ensuring consistency and portability across different environments.

Orchestration platforms, such as Kubernetes, have emerged as the standard for managing containerized workloads in production environments [Sayfan 2019]. Kubernetes provides robust features for automating deployment, scaling, and management of microservice-based applications, enabling organizations to achieve high availability, resilience, and scalability.

3. Related Work

This section reviews existing research in microservices security, with a particular focus on authentication and authorization mechanisms, as well as performance assessment techniques in microservices environments. The objective is to position this study in the context of prior work and highlight the research gap that motivates our contribution.

3.1. Microservices Security

Nasab et al. [Nasab et al. 2023] conducted an empirical study to identify and validate security practices in microservices systems. Their analysis involved manually reviewing 861 microservices security points from GitHub and Stack Overflow, followed by a survey of 74 practitioners to assess the practical relevance of these practices. In the context of authentication and authorization, the study found that the most established mechanisms fall into two main categories: centralized and decentralized approaches.

Pereira-Vale et al. [Pereira-Vale et al. 2019] performed a similar mapping study to identify the security mechanisms used in microservice-based systems described in the literature. The study selected 26 articles and applied a rigorous protocol to extract, classify, and organize them. According to the study, the security aspects in microservice-based systems that are considered most important are **Authentication**, **Authorization** and **Credentials**, with their findings in regard to commonly used solutions also relying on centralized and decentralized mechanisms.

3.2. Mechanisms for Authentication and Authorization

Triartono and Negara [Triartono et al. 2019] proposed implementing *Role-Based Access Control* (RBAC) within OAuth 2.0 to enhance authentication and authorization in backend microservices architectures. Their approach, developed using the *Laravel Framework* and *Laravel Passport Library*, resulted in a more secure access control mechanism. This solution follows a centralized security model, providing a unified point of control.

Yarygina and Bagge [Yarygina and Bagge 2018] proposed securing internal service-to-service communication using *Mutual Transport Layer Security* alongside a self-hosted *Public Key Infrastructure*. Their work also explores the use of tokens and local authentication as emerging trends in microservices security. The security mechanisms discussed are primarily decentralized, aligning with the second design principle of distributed systems: individual nodes should make decisions based on locally available information, promoting loose coupling.

3.3. Microservices Performance Assessment

Sedghpour and Townend [Sedghpour and Townend 2022] proposed using eBPF (*extended Berkeley Packet Filter*) as an efficient way to measure the performance of microservices. The eBPF is a technology that allows for dynamic tracing and monitoring of events within the Linux kernel. It provides a flexible way to analyze and manipulate network packets, system calls, and other events in real time. They argued that this technology fits into measuring the performance of microservices by enabling deep visibility into system performance characteristics at the kernel level.

Miano et al. [Miano et al. 2021] conducted research on the use of eBPF-based network functions in microservices. In their work, they demonstrated how eBPF programs can be used to monitor network traffic, measure latency, and monitor resource utilization in microservices environments.

Sedghpour, Klein and Tordsson [Sedghpour et al. 2021] evaluated the performance impact of the **Retry** and **Circuit Breaker** patterns as implemented by two popular open-source resilience libraries: *Polly* for C# and *Resilience4j* for Java. The evaluation results showed that the Retry pattern could be more effective than the Circuit Breaker pattern in reducing contention for external resources.

Costa et al. [Costa et al. 2022] performed an empirical study to reveal the impact of different resiliency patterns, including circuit breaking and retry mechanisms, in various scenarios. They explored the performance implications of these patterns and proposed an adaptive service mesh controller in order to improve performance.

Despite operating within the realm of microservices, these studies focused on fault tolerance and have not specifically examined security patterns.

3.4. Research Gap

While existing literature offers valuable contributions to understanding microservices security practices and general performance assessment techniques, there is a clear lack of empirical studies that focus specifically on the performance implications of authentication and authorization patterns in microservices environments.

4. Performance Comparison

4.1. Host Application

The cornerstone of this study is a microservice-based application [Cardoso 2024], developed specifically to enable this performance comparison, written using the Go programming language. The choice of Go stems from its reputation as a high-performance language, well-suited for microservices applications. The chosen approach for inter-service communication consists of employing HTTP requests, as it aligns with prevalent practices in the field. This methodology seeks to encapsulate the essence of microservices applications in a representative model.

The application simulates a banking system's functionalities, including user and account management, transaction processing, and notification services, comprising five distinct microservices. The choice of five microservices was strategically made to represent a mid-size call chain, allowing the observation of up to five internal network hops.

This setup is sufficient to capture the cumulative performance degradation and resource overhead introduced by security patterns without the interference of excessive network noise that could mask the core experimental results. At the forefront of the application lies a gateway, orchestrating incoming requests and redirecting them to the appropriate services.

The application's API consolidates individual service endpoints. While some requests are handled directly by the respective services, others necessitate inter-service communication, engaging up to four services. Table 1 shows the endpoints provided by the API and their depths (number of internal API calls performed), respectively.

Table 1. Endpoint Depths and Involved Services

Endpoint	Involved Services	Depth
`createAccount`	1	0
`addToBalance`	1	0
`subtractFromBalance`	1	0
`createCustomer`	2	1
`transferAmount`	2	2
`notify`	3	2
`transferAmountAndNotify`	4	5

The services are encapsulated within individual Docker containers, ensuring isolation and portability across various deployment environments. Each service is paired with a dedicated PostgreSQL database instance, ensuring encapsulated processing and supporting loose coupling. Operations within the services consist of standard CRUD (**C**reate, **R**ead, **U**pdate, **D**elete) actions, performed on their respective databases.

4.2. Employed Authentication and Authorization Mechanism

To support the implementation of different application versions, a basic authentication and authorization method was devised. The core component of this mechanism is a users table, whose schema is shown in Table 2. Each user entry includes a unique user ID, an associated password, and boolean flags indicating the user's permissions.

For each API call within the application, the following process is enforced:

- **Authentication:** The user ID and password provided in the API call are validated against the corresponding entry in the users table. Authentication is considered successful when the provided credentials match a record in the database.
- **Authorization:** After successful authentication, the application checks whether the user has the necessary permissions to perform the requested operation. Table 3 outlines the permissions associated with each API endpoint.

It is important to note that the complexity of this mechanism is not the focus of this study. Our primary goal is to compare the performance of different patterns, rather than to evaluate the absolute metric values associated with each pattern.

Table 2. Users Table Schema

Field Name	Data Type	Constraints
`id`	VARCHAR(255)	Primary Key
`password`	VARCHAR(255)	Not Null
`can_create`	BOOLEAN	Not Null
`can_deposit`	BOOLEAN	Not Null
`can_withdraw`	BOOLEAN	Not Null
`can_notify`	BOOLEAN	Not Null

Table 3. Endpoint Required Permissions

Endpoint	Required Permissions
`createAccount`	Create
`createCustomer`	Create
`addToBalance`	Deposit
`subtractFromBalance`	Withdraw
`transferAmount`	Deposit, Withdraw
`notify`	Notify
`transferAmountAndNotify`	Deposit, Withdraw, Notify

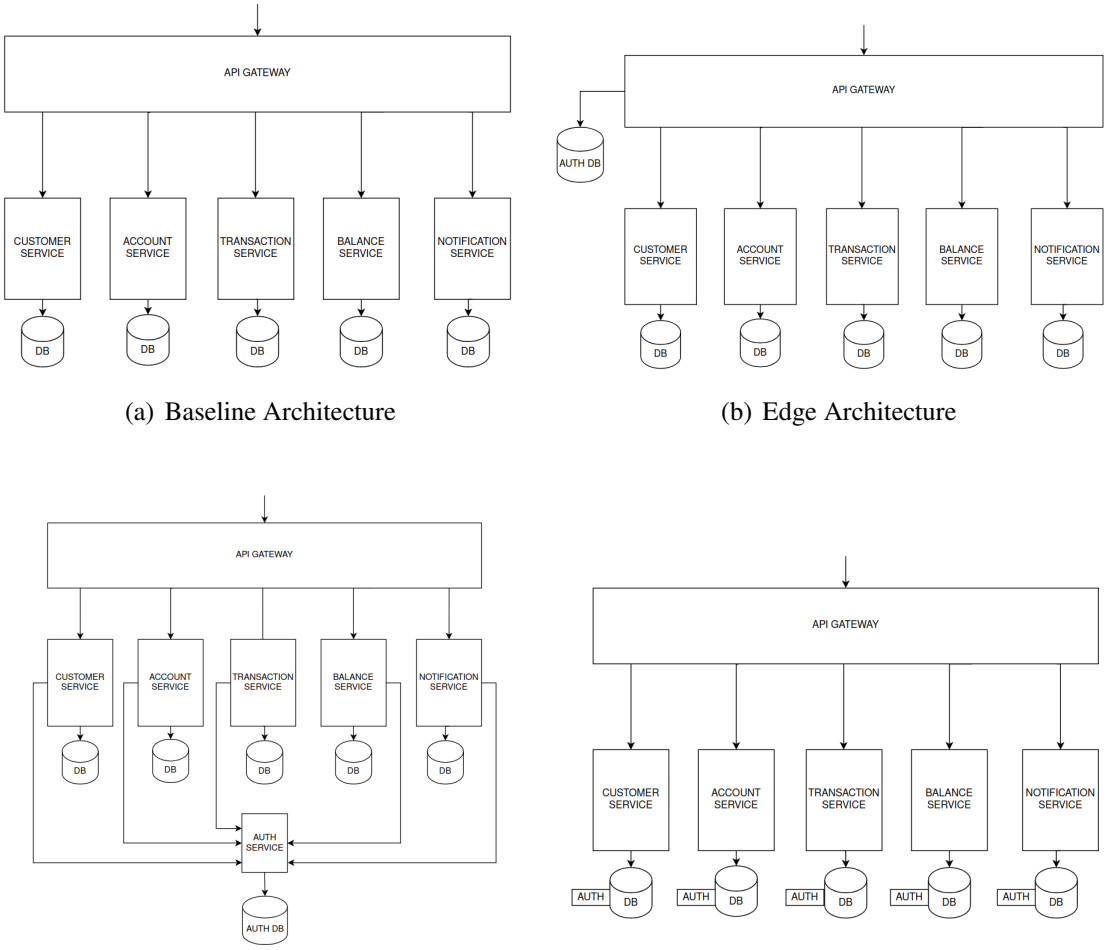
4.3. Application Versions

Four distinct versions of the application were developed to examine various approaches to authentication and authorization within the microservices architecture. These versions facilitate a comparative analysis of patterns and their impact on application performance. The architectures for each version are presented in Figure 1.

- **Baseline Version:** The basic application, which does not include any authentication or authorization mechanisms. Requests are processed without validation of user identity or permissions.
- **Edge Version:** Authentication and authorization are handled at the gateway level, before requests are forwarded to the respective services.
- **Centralized Version:** A dedicated authentication and authorization service is introduced. In this version, all services interact with the central service to verify user identities and enforce access controls.
- **Decentralized Version:** Each service independently handles authentication and authorization using its own database.

4.4. Deploy Environment

All versions of the application were deployed within a Kubernetes cluster, with all components isolated in a dedicated namespace. Within this environment, each of the five services (and the additional authentication service in the centralized pattern) ran as independent pods. Dedicated pods were also provisioned for each service's database and for the application gateway. Each pod was allocated a dedicated CPU core and 1 GB of memory, with strict resource limits enforced to ensure consistency across all deployments.



(a) Baseline Architecture (b) Edge Architecture

(c) Centralized Architecture (d) Decentralized Architecture

Figure 1. Comparative view of the four application architectures evaluated.

To ensure accurate monitoring with minimal overhead, the namespace utilized Prometheus [Cloud Native Computing Foundation 2014] and eBPF Hubble [Cillium 2021], both of which are industry-standard tools for high-fidelity observability.

4.5. Performed Experiments

The evaluation experiments were conducted using a custom Python tool designed to interface with the application gateway, employing a black-box testing approach.

A systematic load test was executed for each API endpoint, issuing 20,000 sequential requests to assess system performance. To enhance statistical robustness, the test was repeated ten times, resulting in a total of 200,000 requests per endpoint.

Data collection was performed at two levels. Externally, the Python tool recorded temporal metrics, simulating user-level interactions with the system. Internally, the monitoring tools integrated into the namespace captured resource consumption and network usage metrics. This dual-level data collection approach enabled a comprehensive assessment of system behavior from both operational and user perspectives.

This evaluation framework was applied consistently across all application versions, ensuring a fair and detailed comparison of the authentication and authorization patterns under study. Figure 2 depicts the metrics collection architecture employed in the experiments.

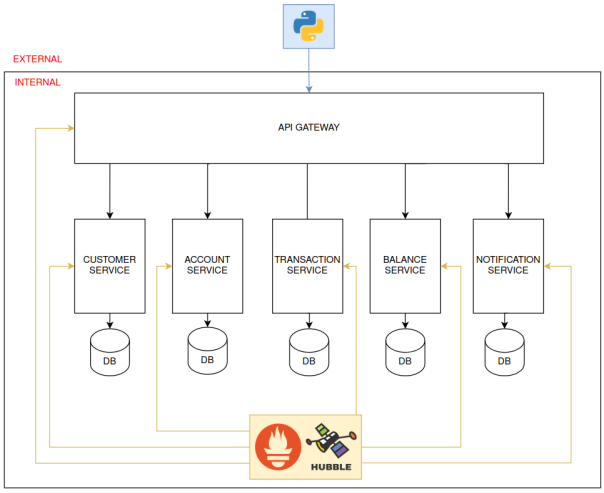


Figure 2. Metrics collection architecture.

5. Results and Discussion

In selecting endpoints for presenting the results and performing analysis, four endpoints were strategically chosen to span depths 0 (*createAccount*), 1 (*createCustomer*), 2 (*transferAmount*), and 5 (*transferAmountAndNotify*) within the application architecture, offering insights into varying levels of service interactions. By concentrating on a subset of endpoints that emulate real-world interactions, we streamline the analysis process while maintaining the potential for extrapolating findings to the broader endpoint set.

Table 4. Mean Response Time

Depth	Endpoint (Version)	Time (ms)	SD
0	`createAccount` (Baseline)	4.26	0.01
	`createAccount` (Edge)	5.30 (+24.41%)	0.01
	`createAccount` (Centralized)	5.86 (+37.55%)	0.02
	`createAccount` (Decentralized)	4.90 (+15.02%)	0.02
1	`createCustomer` (Baseline)	5.85	0.01
	`createCustomer` (Edge)	6.98 (+19.31%)	0.01
	`createCustomer` (Centralized)	8.45 (+44.44%)	0.02
	`createCustomer` (Decentralized)	7.39 (+26.32%)	0.02
2	`transferAmount` (Baseline)	7.18	0.01
	`transferAmount` (Edge)	8.58 (+19.49%)	0.01
	`transferAmount` (Centralized)	10.74 (+49.58%)	0.02
	`transferAmount` (Decentralized)	9.07 (+26.32%)	0.02
5	`transferAmountAndNotify` (Baseline)	10.39	0.02
	`transferAmountAndNotify` (Edge)	11.97 (+15.20%)	0.02
	`transferAmountAndNotify` (Centralized)	16.78 (+61.50%)	0.03
	`transferAmountAndNotify` (Decentralized)	14.06 (+35.32%)	0.02

Table 5. Mean Requests Per Second (Throughput)

Depth	Endpoint (Version)	Requests Per Second
0	`createAccount` (Baseline)	228.78
	`createAccount` (Edge)	184.07 (80.45%)
	`createAccount` (Centralized)	167.31 (73.13%)
	`createAccount` (Decentralized)	199.61 (87.24%)
1	`createCustomer` (Baseline)	169.48
	`createCustomer` (Edge)	141.98 (83.77%)
	`createCustomer` (Centralized)	117.51 (69.33%)
	`createCustomer` (Decentralized)	134.29 (79.23%)
2	`transferAmount` (Baseline)	138.51
	`transferAmount` (Edge)	115.89 (83.66%)
	`transferAmount` (Centralized)	92.63 (66.87%)
	`transferAmount` (Decentralized)	109.69 (79.19%)
5	`transferAmountAndNotify` (Baseline)	95.83
	`transferAmountAndNotify` (Edge)	83.14 (86.75%)
	`transferAmountAndNotify` (Centralized)	59.38 (61.96%)
	`transferAmountAndNotify` (Decentralized)	70.85 (73.93%)

5.1. Performance

Tables 4 and 5 present mean performance metrics for the key endpoints across the four versions of the application: baseline, edge, centralized, and decentralized. Each metric is expressed both in terms of response time (in milliseconds) and requests per second. The metrics for the versions that perform authentication and authorization contain a percentage indicator in terms of the baseline version's obtained results. This indicator is consistently used in all subsequent tables.

Overall, the centralized version consistently demonstrates the highest increase in response time and the lowest performance across all endpoints compared to the baseline. The edge and decentralized versions show comparatively lower increases in response time and maintain better performance levels.

5.2. Resource Consumption

Table 6. Mean Total CPU Usage

Depth	Endpoint (Version)	CPU Usage (%)	SD
0	`createAccount` (Baseline)	46.54	0.38
	`createAccount` (Edge)	54.46 (+17.01%)	0.46
	`createAccount` (Centralized)	61.15 (+31.39%)	0.49
	`createAccount` (Decentralized)	53.51 (+14.97%)	0.44
1	`createCustomer` (Baseline)	48.41	0.39
	`createCustomer` (Edge)	53.25 (+09.99%)	0.44
	`createCustomer` (Centralized)	63.66 (+31.50%)	0.50
	`createCustomer` (Decentralized)	55.31 (+14.25%)	0.45
2	`transferAmount` (Baseline)	49.92	0.40
	`transferAmount` (Edge)	55.23 (+10.63%)	0.43
	`transferAmount` (Centralized)	63.79 (+27.78%)	0.49
	`transferAmount` (Decentralized)	56.01 (+12.20%)	0.46
5	`transferAmountAndNotify` (Baseline)	52.83	0.41
	`transferAmountAndNotify` (Edge)	58.63 (+10.97%)	0.45
	`transferAmountAndNotify` (Centralized)	67.36 (+27.50%)	0.49
	`transferAmountAndNotify` (Decentralized)	59.17 (+12.00%)	0.45

Tables 6 and 7 present mean resource consumption metrics for the key endpoints across the four versions of the application, aggregating each service's metrics in order to provide a total application amount. Resource consumption is measured in terms of total CPU usage (expressed as a percentage of 1 dedicated CPU) and total memory usage (expressed in MiB).

The centralized version consistently demonstrates elevated total CPU usage and memory consumption compared to the other configurations, indicating it places a heavier demand on system resources. Conversely, the edge and decentralized versions exhibit moderate rises in resource consumption, suggesting that they are more balanced approaches in terms of resource utilization.

Table 7. Mean Total Memory Usage

Depth	Endpoint (Version)	M. Usage (MiB)	SD
0	`createAccount` (Baseline)	171.74	0.12
	`createAccount` (Edge)	205.67 (+19.75%)	0.17
	`createAccount` (Centralized)	214.82 (+25.08%)	0.18
	`createAccount` (Decentralized)	210.41 (+22.51%)	0.18
1	`createCustomer` (Baseline)	173.84	0.13
	`createCustomer` (Edge)	206.68 (+18.89%)	0.17
	`createCustomer` (Centralized)	216.98 (+24.81%)	0.18
	`createCustomer` (Decentralized)	211.29 (+21.54%)	0.18
2	`transferAmount` (Baseline)	175.69	0.12
	`transferAmount` (Edge)	209.37 (+19.17%)	0.17
	`transferAmount` (Centralized)	218.49 (+24.36%)	0.18
	`transferAmount` (Decentralized)	213.68 (+21.62%)	0.18
5	`transferAmountAndNotify` (Baseline)	177.41	0.12
	`transferAmountAndNotify` (Edge)	210.96 (+18.91%)	0.17
	`transferAmountAndNotify` (Centralized)	219.99 (+24.00%)	0.19
	`transferAmountAndNotify` (Decentralized)	215.21 (+21.30%)	0.18

5.3. Network Usage

Table 8. Mean Total Transmitted Data

Depth	Endpoint (Version)	Data (MB/s)	SD
0	`createAccount` (Baseline)	1.12	0.01
	`createAccount` (Edge)	1.26 (+12.50%)	0.01
	`createAccount` (Centralized)	1.46 (+30.35%)	0.10
	`createAccount` (Decentralized)	1.37 (+22.32%)	0.05
1	`createCustomer` (Baseline)	1.26	0.02
	`createCustomer` (Edge)	1.44 (+14.28%)	0.03
	`createCustomer` (Centralized)	1.58 (+25.39%)	0.09
	`createCustomer` (Decentralized)	1.55 (+23.01%)	0.08
2	`transferAmount` (Baseline)	1.30	0.02
	`transferAmount` (Edge)	1.52 (+16.92%)	0.05
	`transferAmount` (Centralized)	1.69 (+30.00%)	0.12
	`transferAmount` (Decentralized)	1.62 (+24.61%)	0.10
5	`transferAmountAndNotify` (Baseline)	1.44	0.03
	`transferAmountAndNotify` (Edge)	1.64 (+13.88%)	0.06
	`transferAmountAndNotify` (Centralized)	1.92 (+33.33%)	0.15
	`transferAmountAndNotify` (Decentralized)	1.79 (+24.30%)	0.11

Table 8 provides insights into the mean network usage across the key endpoints for the four versions of the application. Network usage is measured in terms of total transmitted data in megabytes per second (MB/s).

The centralized version consistently records the highest total transmitted data across all endpoints compared to the baseline configuration. This suggests that centralized mechanisms contribute to elevated network usage compared to both edge and decentralized approaches.

5.4. Disk Usage

Table 9. Total Application Disk Usage

Application Version	Total Disk Usage (MB)
Baseline	242.90
Edge	322.00 (+32.56%)
Centralized	322.00 (+32.56%)
Decentralized	394.00 (+62.20%)

Table 9 illustrates the total disk space utilized by different versions of the application, reflecting the storage requirements of user data across the experiment. In comparison to the baseline, all versions demonstrate an increase in disk usage. Both the edge and centralized versions exhibit identical disk usage levels, amounting to 132.56% of the baseline. Conversely, the decentralized version displays the highest disk usage, accounting for 162.20% of the baseline.

The observed disparity in disk usage across the versions can be attributed to the underlying architecture of the authentication and authorization mechanisms. In the edge and centralized versions, user data is centralized within a single database instance, resulting in a more efficient storage utilization model. However, the decentralized version necessitates the replication of user data across multiple service databases, contributing to a higher disk space consumption. Scaling from 5 to 100 services would result in a linear growth of storage requirements ($\mathcal{O}(n)$) for the decentralized pattern, potentially leading to a 1000% overhead compared to the baseline, which poses a significant challenge for massive-scale deployments.

6. Conclusions

The conclusions drawn from this study shed light on the intricate dynamics of securing microservices architectures. The host application and the deployment environment were designed to be very similar to real-world microservices use cases, enabling us to extrapolate the results obtained to a larger scope.

The collected data showed that the choice of pattern significantly influences the application profile. The centralized approach consistently exhibited the highest response times and resource utilization across all endpoints, indicating potential scalability and efficiency challenges. Conversely, the edge and decentralized approaches emerged as promising alternatives, displaying better outcomes. In contrast, the decentralized version, which necessitates multiple copies of user data distributed across service databases, incurs substantially higher disk usage.

These findings underscore the importance of carefully considering different authentication and authorization strategies. While centralized mechanisms may offer administrative convenience and a diminished need for storage space, they come at the cost of increased resource utilization and reduced system performance. On the other hand, decentralized approaches offer better performance and network efficiency, but increase disk usage and complexity in implementation and management.

The implementation followed a direct approach based on the definitions of each pattern as outlined in the literature. However, various optimizations could be made to enhance performance. For example, in the centralized pattern, replicating the authentication service across multiple instances could alleviate some of the scalability and response time challenges. Similarly, in the decentralized model, sharing authentication data between services could reduce disk usage but at the cost of also reducing the independence of individual services. Other optimizations, such as caching user data or implementing hybrid approaches, could further improve performance while still maintaining key benefits of each pattern.

Although the results point to relatively well-known trade-offs between centralized and decentralized patterns in other areas of computing, it is important to highlight that no study has been carried out supporting these conclusions in this specific field.

It is also imperative to acknowledge the security implications inherent in the choice of authentication and authorization mechanisms. Unlike the centralized and decentralized approaches, which offer service-level authentication and authorization, the edge version operates only at the edge level. This enables vulnerabilities in terms of security, as it does not contemplate defense-in-depth. In security-critical environments or those adhering to zero-trust principles, the benefits of service-level approaches outweigh the performance gains offered by the edge paradigm. As such, this decision should be aligned with the organization's security posture, regulatory requirements, and risk tolerance levels.

Future work will explore these authentication patterns in asynchronous, event-driven environments (e.g., using message brokers like Kafka or RabbitMQ), where token propagation and validation dynamics differ significantly from the synchronous HTTP-based communication analyzed in this study.

References

Cardoso, R. (2024). *Microservices Auth Benchmark*. https://github.com/rfcardoso07/microservices-auth-benchmark.

Cillium (2021). *Hubble*. https://github.com/cilium/hubble.

Cloud Native Computing Foundation (2014). *Prometheus*. https://prometheus.io.

Costa, T., Vasconcelos, D., Aderaldo, C., and Mendonça, N. (2022). Avaliação de desempenho de dois padrões de resiliência para microsserviços: Retry e circuit breaker. In *Anais do XL Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 517–530, Porto Alegre, RS, Brasil. SBC.

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

Fernando, R. and Wickramaarachchi, D. (2022). Performance optimization of microservice applications under resource constrained environments. In *2022 International Research Conference on Smart Computing and Systems Engineering (SCSE)*, volume 5, pages 309–313.

Fowler, M. (2014). *Microservices: a definition of this new architectural term*. https://martinfowler.com/articles/microservices.html.

Guerrero, C., Lera, I., and Juiz, C. (2018). Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications. *The Journal of Supercomputing*, 74(7):2956–2983.

Heinrich, R., Van Hoorn, A., Knoche, H., Li, F., Lwakatare, L. E., Pahl, C., Schulte, S., and Wettinger, J. (2017). Performance engineering for microservices: research challenges and directions. In *Proceedings of the 8th ACM/SPEC on international conference on performance engineering companion*, pages 223–226.

Miano, S., Risso, F., Bernal, M. V., Bertrone, M., and Lu, Y. (2021). A framework for ebpf-based network functions in an era of microservices. *IEEE Transactions on Network and Service Management*, 18(1):133–151.

Nasab, A. R., Shahin, M., Raviz, S. A. H., Liang, P., Mashmool, A., and Lenarduzzi, V. (2023). An empirical study of security practices for microservices systems. *Journal of Systems and Software*, 198:111563.

Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc.

OWASP (2017). *Microservices Security Cheat Sheet*. https://cheatsheetseries.owasp.org/cheatsheets/Microservices_Security_Cheat_Sheet.html.

Pereira-Vale, A., Márquez, G., Astudillo, H., and Fernandez, E. B. (2019). Security mechanisms used in microservices-based systems: A systematic mapping. In *2019 XLV Latin American Computing Conference (CLEI)*, pages 01–10.

Sayfan, G. (2019). *Hands-On Microservices with Kubernetes: Build, deploy, and manage scalable microservices on Kubernetes*. Packt Publishing Ltd.

Sedghpour, M. R. S., Klein, C., and Tordsson, J. (2021). Service mesh circuit breaker: From panic button to performance management tool. In *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems*, HAOC '21, page 4–10, New York, NY, USA. Association for Computing Machinery.

Sedghpour, M. R. S. and Townend, P. (2022). Service mesh and ebpf-powered microservices: A survey and future directions. In *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 176–184.

Triartono, Z., Negara, R. M., and Sussi (2019). Implementation of role-based access control on oauth 2.0 as authentication and authorization system. In *2019 6th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, pages 259–263.

Yarygina, T. and Bagge, A. H. (2018). Overcoming security challenges in microservice architectures. In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 11–20. IEEE.