Internet of Things 36 (2026) 101867

Contents lists available at ScienceDirect

Internet of Things

journal homepage: www.elsevier.com/locate/iot

Analysis of microservices-based IoT systems: deployment challenges, industry practices, and performance insights

Yahia El Fellah [a,*], Jean Baptiste Minani [b], Naouel Moha [a], Julien Gascon-Samson [a], Yann-Gaël Guéhéneuc [b]

[a] École de Technologie Supérieure, 1100 Notre-Dame St W, Montreal, H3C 1K3, QC, Canada
[b] Concordia University, 1455 Blvd. De Maisonneuve Ouest, Montreal, H3G 1M8, QC, Canada

ARTICLE INFO

Keywords:
Microservices
IoT system design
IoT system deployment
SE practices
Deployment challenges
Resource utilization
Performance optimization

ABSTRACT

As the adoption of microservices in Internet of Things (IoT) systems grows, deploying them on the Edge remains a significant challenge for practitioners. While Edge Computing offers improved latency and resource efficiency by processing data near the source, it also introduces complexity in managing microservices. Despite increasing academic interest, few comprehensive studies have investigated the specific challenges and effective software engineering (SE) practices for deploying microservices-based IoT systems on the Edge. Therefore, we conducted a multi-method study to bridge this gap. We used three methods: (1) a systematic literature review (SLR) to identify known challenges and approaches, (2) a gray literature review (GLR) to extract SE practices used in the field, and (3) an empirical evaluation using two versions of a case study, one with and one without selected SE practices. The findings show that (1) the most reported challenges relate to resource utilization and performance optimization, (2) containerized microservices, API gateways, and database-per-service are among the most commonly recommended practices, and (3) implementing these practices led to a 132% throughput improvement, 49% reduction in latency, and memory savings of up to 13% in Edge-based IoT systems. However, increased architectural complexity also led to higher CPU usage. This study offers a catalog of best practices and empirical evidence to support IoT developers aiming to optimize microservices-based deployments on the Edge, particularly in resource-constrained environments.

1. Introduction

The Internet of Things (IoT) is a concept where devices, software, and network technologies are interconnected to exchange data with various devices and systems via the Internet [1–3]. The IoT interconnects the physical world with computer networks through technologies such as sensors, RFID, Bluetooth, and embedded systems, allowing objects to be detected, controlled, automated, and analyzed, and creating opportunities for advanced efficiency and integration across many sectors and applications [4–6]. As the number of connected devices increases [7], it becomes challenging for IoT systems to manage and process IoT data streams [8,9], emphasizing the critical need for efficient data processing and analysis due to the volume, velocity, and variety of data. In this paper, the term IoT system refers to the end-to-end stack consisting of connected devices, sensors, actuators, gateways, edge nodes, and cloud services that exchange data over the Internet. Our study focuses on SE practices for the microservices application deployed at the edge.

* Corresponding author.
E-mail addresses: delyahia@gmail.com (Y. El Fellah), naouel.moha@etsmtl.ca (N. Moha), julien.gascon-samson@etsmtl.ca (J. Gascon-Samson), yann-gael.gueheneuc@concordia.ca (Y.-G. Guéhéneuc).

https://doi.org/10.1016/j.iot.2025.101867
Received 23 June 2025; Received in revised form 5 December 2025; Accepted 31 December 2025
Available online 7 January 2026

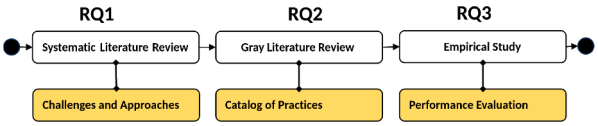


Fig. 1. Multi-faceted study approach.

Device firmware and low level drivers are not considered in this study. The data collected from various devices grows exponentially to the extent that traditional cloud computing approaches, which rely on sending all data to centralized data centers, can become inadequate. These approaches can lead to latency issues, which is unacceptable for scenarios requiring real-time decision-making such as healthcare monitoring systems, where patient data must be acted on without delay. Edge Computing complements IoT by enabling faster data processing and analysis at the edge of the network. The need for real-time and efficient processing has led to the adoption of Edge computing for IoT systems, which enables data distribution and processing closer to devices for real-time analysis. Beyond latency, proximity also reduces backbone bandwidth via local filtering/aggregation, supports data sovereignty by keeping sensitive data on device/site or within jurisdiction, tolerates intermittent connectivity through local operation with deferred sync, and improves localized fault isolation and graceful degradation. However, Edge resources can be limited [10], which requires careful optimization to handle the high data volume, while meeting real-time processing needs for critical applications [11]. Microservice is a software architectural style that structures an application as a collection of small, loosely coupled services. This architectural style can be adapted to deploy IoT systems on the Edge. In IoT and edge systems, the boundaries and placement of microservices are primarily determined by device and gateway constraints, as well as proximity to data sources, rather than by team or organizational structures, as is the case in enterprise microservices.

The microservices architecture gained attention for its ability to efficiently tackle the complexities of IoT systems, both in academia and industry [12–14]. Consequently, microservices have become foundational *building blocks* in many industrial IoT systems and are now a prevalent architecture in IoT system design. This trend has led to numerous studies exploring their integration in IoT systems and their effects [15,16].

While a microservices-based architectural style offers benefits in an IoT context, it raises several challenges due to the significant levels of heterogeneity of resources in IoT systems [17]. In particular, when implementing microservices in resource-constrained Edge environments, there can be several challenges such as computing and resources utilization [18,19]. Consequently, several studies (e.g., [20]) have explored the deployment of microservices at the Edge and have identified many challenges. Overcoming those challenges and realizing the potential benefits of microservices for IoT systems remains an open issue. However, there is a research gap addressing those challenges using SE practices. SE practices are defined as a set of practices or techniques used in SE for a successful execution of software projects and that includes but not limited to design, coding, documentation, deployment, etc. [21,22]. These practices not only help with the execution of the software projects but also about ensuring their quality, maintainability, and efficiency. Among the SE practices, “containerization of microservices” has gained significant attention in the IoT systems. Containerization involves encapsulating a microservice in a container with its own runtime environment. A number of studies (e.g., [23–26]) investigated how this practice is applied in IoT contexts. These studies specifically examine how containerization can improve the management of resources, especially on the Edge where resource constraints are common. However, while there is promise in utilizing SE practices to address those challenges, the existing literature remains conceptual or only validates single practices through limited case studies [27]. There is a notable research gap in comprehensive empirical evidence demonstrating the performance benefits of an integrated set of SE practices for microservices-based IoT systems. In our study, we aim to identify the challenges related to deploying microservices-based IoT systems at the Edge. Additionally, we seek to identify SE best practices to address these challenges and assess the impact of implementing selected practices on the deployment of microservices-based IoT systems at the Edge. Our assessment focuses on two key performance metrics: throughput and latency, while also evaluating CPU and memory usage to measure resource utilization. Beyond performance, microservice granularity serves as an architectural lever that affects scalability (elasticity and load isolation), evolvability and modular refactoring (change isolation), and fault tolerance (failure containment). In this study, we limit our analysis to performance impacts and will address these other dimensions in future work. We conducted a multi-faceted study consisting of three main steps. Each step aims to answer one research question (RQ), as illustrated in Fig. 1 and described in the following.

RQ1: What are the challenges faced by IoT system developers in deploying microservices-based IoT systems on the Edge?

We conducted an SLR to identify challenges faced by IoT system developers when deploying microservice-based IoT systems, as well as approaches to mitigate those challenges. As a result, we found several challenges mostly related to resource utilization and performance optimization. We focused on SE practices that could address these challenges. However, we did not find SE practices reported in the SLR that directly address these challenges in microservices-based IoT systems at the Edge. We complemented the SLR with a Gray Literature Review (GLR) to explore industry-practiced SE techniques that were not prominent in the academic sources identified. Because the microservices/Edge space evolves quickly, relevant practices may first appear in engineering blogs, vendor white papers, and OSS repositories. Thus, we conducted a GLR to identify potential SE practices that could optimize performance and resource utilization.

RQ2: What SE practices could optimize the performance and resource utilization of microservices-based IoT systems on the Edge?

We conducted a GLR to identify SE practices that can help improve the performance of microservices-based IoT systems. As a result, we compiled a catalog of SE practices reported in GLR improving microservices-based system performance and resource utilization.

The catalogue contains some practices such as the containerization of microservices, and the use of the database-per-service pattern. By identifying and compiling a catalog of relevant SE practices, we established a knowledge base to inform our empirical study.

RQ3: What is the performance impact of some of the identified SE practices for microservices-based IoT systems deployed on the Edge?

We conducted an empirical study to evaluate the performance impact of selected SE practices on a microservices-based IoT system. Specifically, we used a proof-of-concept system called WIMP that helps students determine their professor's availability. This empirical study involves implementing WIMP using a microservices architecture and adhering to identified SE practices from our catalog. This implementation serves as a valuable resource for researchers to conduct experiments on microservices-based IoT systems. We then empirically compared the performance of WIMP with and without the selected SE practices. The results showed the performance improvements achieved by applying a subset of SE practices, such as reduced latency and memory usage, in a concrete IoT system scenario.

The rest of this paper is organized as follows: Section 2 focuses on related works. The SLR process and findings are described in Section 3. The GLR process and findings are shown in Section 4. Section 5 presents the empirical study design and the performance evaluation results. Section 6 presents the possible threats that could affect the validation of our study. Finally, Section 7 concludes and presents future work.

2. Related work

In this section, we review related work and organize it into three categories: integration of microservices architecture within IoT, performance evaluation of microservices in IoT environments, and optimization strategies for microservices architectures in IoT. We acknowledge broader microservices studies that are not specific to IoT, but exclude them to focus on IoT specific studies.

Deployment challenges in IoT: Microservices architecture has emerged as a popular approach for developing and deploying IoT systems [19,28]. Previous studies explored the deployment of microservices in IoT contexts (e.g., [26]). Existing studies [29,30] examined architectural frameworks and methodologies for creating and deploying IoT systems, presenting strategic directions for introducing microservices into IoT ecosystems. Furthermore, studies such as Aksakalli and Can [31] outlined communication and deployment methodologies for microservices and highlighted related challenges through SLR. Taken together, these studies primarily discuss deployment challenges in a general IoT context.

SE practices in IoT: As the IoT ecosystem expands, especially on the Edge, there is growing interest in how SE practices help address deployment challenges. In this paper, we distinguish between *containerization* (a deployment mechanism for packaging and isolating services) and *service self containment* (a design principle emphasizing cohesive functionality and minimal shared state). Previous studies investigated the use of container based virtualization in IoT systems [23–26,32,33] and highlighted the importance of tailoring microservices implementations and container orchestration strategies to the demands of Edge Computing. Furthermore, one study [26] investigated patterns such as service self containment (i.e, internal coherence and minimal shared state) and orchestration (i.e., coordination across services) that can be adopted by IoT systems. These works discuss selected practices rather than a consolidated catalog targeting edge specific deployment challenges, and they mainly provide conceptual discussions and recommendations. Our study aims to complement this body of work by compiling a catalog and empirically examining a combined subset of practices in an edge setting.

SE practices evaluation in IoT: Several studies evaluated aspects of microservices architecture in IoT systems. Authors in Lira et al. [17] assessed the performance of IoT systems using a reactive microservices based architecture under various deployment policy scenarios at the Edge. Additionally, authors in Akbulut and Perros [34] analyzed the performance of microservices in cloud deployment scenarios using practices such as the API gateway, chain of responsibility, and asynchronous messaging. This study focuses on three design patterns. Overall, prior work often considers individual practices or presents conceptual approaches; our work is intended to complement these efforts by evaluating a combined approach and reporting empirical results in an edge deployment context.

In summary, existing studies explored deployment challenges of microservices in IoT systems and the potential of SE practices to address these challenges. Building on these contributions, there remains an opportunity for comprehensive empirical evaluation of a combined set of SE practices-spanning deployment mechanisms (e.g., containerization) and design principles (e.g., service self containment)-for microservices based IoT systems at the Edge; our study addresses this opportunity.

3. Systematic literature review

In this section, we address **RQ1: What are the challenges faced by IoT system developers in deploying microservices-based IoT systems on the Edge?**

We conducted SLR to identify the key challenges encountered by IoT system developers when deploying microservices-based IoT systems on the Edge.

3.1. Review process

We conducted an SLR to understand the current deployment practices and identify the prevalent challenges associated with microservices-based IoT systems. This SLR is guided by the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) [35], which provides a comprehensive framework for conducting and reporting systematic reviews and meta-analyses. The key steps of our literature review process are described in Fig. 2.

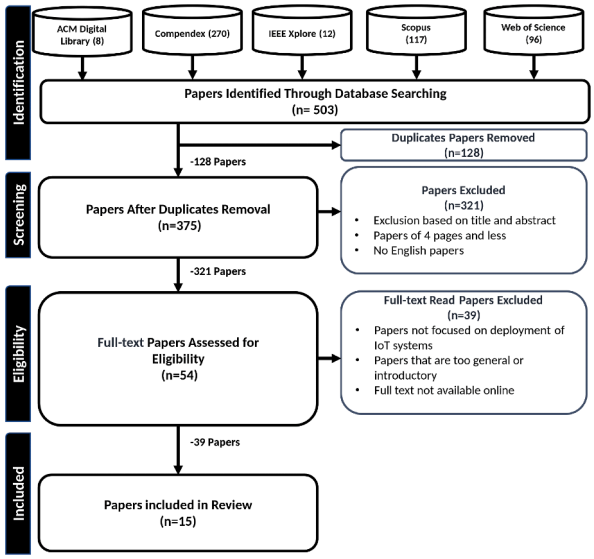


Fig. 2. Steps in literature review.

(deploy OR install OR orchestration) AND (IoT OR Edge OR 'Internet of Things') AND (microservice)

Table 1
Search strings.

Database	Search String	Papers
Compendex	(deploy OR install OR orchestration) AND (IoT OR Edge OR 'Internet of Things') AND (microservice)	270
Scopus	Title((deploy OR install OR orchestration) AND (IoT OR Edge OR 'Internet of Things') AND (microservice))	117
WoS	TI = (deploy OR install OR orchestration) AND TI = (IoT OR edge OR 'Internet of Things') AND TI = (microservice)	96
IEEE Xplore	("Document Title":deploy OR install) AND ("Document Title":IoT OR Edge OR 'Internet of Things') AND ("Document Title":microservice)	12
ACM Digital Library	Title:(deploy OR install OR orchestration) AND"Title:(IoT OR Edge OR 'Internet of Things') AND Title:(microservice)"	8

[➥] **WoS**:Web of Science [➥] **TL**:Title Search Field Tags for WoS

3.1.1. Information sources and search query

We formulated a search query to retrieve scientific papers from digital libraries following the PICO (Population, Intervention, Comparison, Outcome) [36]. Keywords are identified by focusing on terms related to the essence of our research focus, revolving around "microservices","deployment", "challenges", and "IoT". To apply PICO, we performed the following steps:

1. Identifying key terms from the research problem;
2. Identifying alternative terms for key terms in the previous step;
3. Applying the boolean OR between key terms and their alternatives;
4. Applying the boolean AND to combine expressions in the previous step.

As a result, we formulated the following search query.

We customized the search query based on the target the database as listed in Table 1.

To obtain the set of papers, we used specific search strings and accessed five digital libraries including Compendex, Scopus, Web of Science (WoS), IEEE Xplore, and ACM Digital Library. Through this extensive search, we obtained a total of 503 papers.

3.1.2. Inclusion and exclusion criteria

We applied the following inclusion/exclusion criteria:

- **Inclusion Criteria:**
  - The paper is written in English.

Table 2
Papers included in our literature review.

Code	Title	Year	Publisher	Venue	Ref
P1	Microservices for Reliable Safety-Critical Cellular IoT Systems	2024	IEEE Global Communications Conference	Conference	[37]
P2	AI-Enabled Secure Microservices in Edge Computing: Opportunities and Challenges	2023	IEEE Transactions On Services Computing	Journal	[38]
P3	Application Deployment in Mobile Edge Computing Environment Based on Microservice Chain	2022	2022 IEEE 25th International Conference on CSCWD	Conference	[39]
P4	Cost-aware Deployment of Microservices for IoT Applications in Mobile Edge Computing Environment	2022	IEEE Transactions on Network and Service Management	Journal	[40]
P5	Distributed Redundant Placement for Microservice-based Applications at the Edge	2022	IEEE Transactions on Services Computing	Journal	[41]
P6	Dynamic On-Demand Fog Formation Offering On-the-Fly IoT Service Deployment	2020	IEEE Transactions on Network and Service Management	Journal	[42]
P7	Edge-supported Microservice-based Resource Discovery for Mist Computing	2020	2020 IEEE 11th International Conference on DESSERT	Conference	[43]
P8	Optimal Application Deployment in Resource-Constrained Distributed Edges	2021	IEEE Transactions on Mobile Computing	Journal	[44]
P9	ProScale: Proactive Autoscaling for Microservice With Time-Varying Workload at the Edge	2023	IEEE Transactions on Parallel and Distributed Systems	Journal	[45]
P10	Research on deployment method of edge computing gateway based on microservice architecture	2021	IOP Conference Series: Earth and Environmental Science	Journal	[46]
P11	Resource-Aware Dynamic Service Deployment for Local IoT Edge Computing: Healthcare Use Case	2021	IEEE Access	Journal	[47]
P12	Microservice Orchestration for IoT via Multiobjective Deep Reinforcement Learning	2022	IEEE IoT Journal	Journal	[48]
P13	Towards an Easily Programmable IoT Framework Based on Microservices	2018	Journal of Software	Journal	[49]
P14	Towards cost-effective and robust AI microservice deployment in edge computing environments,	2022	Future Generation Computer Systems	Journal	[50]
P15	IoT Microservice Deployment in Edge-Cloud Hybrid Environment Using RL	2021	IEEE IoT Journal	Journal	[51]

[➥] **DESSERT**:Dependable Systems, Services, and Technologies [➥] **CSCWD**: Computer Supported Cooperative Work in Design

  - The paper has 5 pages or more.
  - The paper focuses on deployment of microservice-based IoT systems on the Edge.
  - The paper provides enough details about microservices-based IoT system deployment challenges.
  - The full text is available online.
- **Exclusion Criteria:**
  - The paper that is not peer-reviewed.
  - The paper that is duplicated.
  - The paper that has 4 pages or less.
  - The paper that is not written in English.
  - The paper that does not provide enough details about microservices-based IoT system deployment challenges.

We applied the first four exclusion criteria to retain 54 papers. We conducted a full-text review of the 54 remaining papers, which allowed us to answer our research question. We selected 15 papers that discuss deployment challenges faced in microservices-based IoT systems on the Edge.

3.1.3. Snowballing

We conducted both backward and forward snowballing and we did not identify any additional paper. Table 2 shows the 15 papers included in our review. These papers are publicly accessible in the replication package available on Zenodo[1] or Ptidej website[2].

[1] https://zenodo.org/records/15127718
[2] https://www.ptidej.net/downloads/replications/iotj25a

Takeaway from RQ1: The challenges in deploying microservices-based IoT systems include placement optimization, network issues, programming complexity, dynamic load balancing, orchestration, and resource management. The literature proposes various algorithms, frameworks, deployment strategies, and learning-based approaches, as well as architectural designs leveraging containerization, APIs, and event-driven models. These solutions can help overcome resource utilization and performance optimization challenges in resource-constrained edge environments.

3.2. Analysis

We conducted a detailed analysis of the 15 papers identified in Section 3.1 to address **RQ1**. We described the challenges of deploying microservices-based IoT systems on the Edge and described the proposed approaches to address these challenges. We identified the following challenges:

Placement Problem Challenges. These challenges concern optimizing the placement of microservices on the most suitable cloud or edge servers to meet optimization objectives like reducing latency. Various techniques have been proposed, including mathematical models [40,42], heuristics [39,41], meta-heuristics based on swarm intelligence [37], evolutionary algorithms [44], and machine learning approaches [39,41].

Dynamic Load Balancing Challenges. These challenges concern distributing tasks among IoT devices efficiently. For example, an approach, known as ProScale, provides an machine learning (ML) based component that uses a rapid simple moving average (SMA) to forecast the workload associated with each microservice regularly [45]. Another approach implements an automatic service that leverages a resource discovery mechanism to enable the efficient on-the-fly deployment of lightweight microservices [47]. Furthermore, a hybrid resource discovery solution for edge computing is proposed, involving the on-demand deployment of resource directories as containerized microservices to edge devices[43].

Network Challenges. These challenges pertain to network connectivity, latency, and throughput. Approaches like edge caching, data compression, efficient service discovery protocols, and network-aware deployment strategies aim to mitigate these challenges. For instance, [52] proposes hierarchical orchestration to optimize network resource utilization across edge, fog, and cloud.

Development Complexity Challenges. Developing decentralised, distributed microservices-based IoT systems introduces complexity compared to monolithic approaches. Authors in [49] proposes a microservices' framework to simplify IoT application development. Other remedies include domain-driven design, reusable libraries, and templates.

Orchestration and resource management challenges. These challenges pertain to the effective management of computational resources at the edge, aiming to optimise performance. Authors in [50] introduced machine learning models, including deep neural networks, to enhance orchestration. Furthermore, the authors in [53], studied a novel concept called V-Edge (Virtual Edge) to address the challenge of efficient resource management. Authors in [52], proposed an approach called ROMA to enhance the optimisation of computing and network resources in edge computing environments while maintaining high-quality performance. Authors in [46], proposed a solution based on microservice architecture and a deployment method based on the "Gaussian mixture model" to optimize the number of deployed gateway devices while ensuring load balancing.

3.3. Conclusion

Most studies concentrate on proposing particular approaches and solutions for the challenges faced, like optimized algorithms or frameworks that target a specific challenge like placement or discovery. However, none of those challenges is fully addressed by existing studies. A common SE practice seen across many proposed solutions is the utilization of containerisation for implementing microservices. However, we did not find a study that applies comprehensively SE best practices across architecture, design, implementation, testing, deployment, and monitoring to optimize microservices-based IoT as a whole. The existing literature does not systematically establish connections between optimizing resource utilization,performance, and other key metrics within edge environments and SE practices. Optimization in edge environments remains an ongoing study.

4. Gray literature review

This section aims to address **RQ2: What software engineering (SE) practices can optimize the performance and resource utilization of microservices-based IoT systems on the edge**. We focused on SE practices and approaches that can improve resource utilization and performance for microservices-based IoT systems deployed on the edge. We did not identify any SE practices in our SLR; therefore, we conducted a GLR comprised of non-peer-reviewed documents such as blogs, and technical reports to identify those practices.

4.1. Review process

We again followed PRISMA guidelines [35] for conducting a systematic review, as depicted in Fig. 3. In the following sections, we used the term "documents" to refer to white papers, blog posts, industrial technical reports, and other non-peer-reviewed materials found on the Internet.

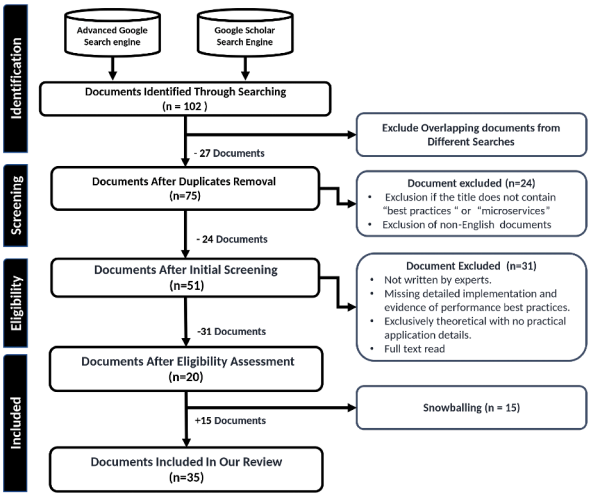


Fig. 3. Steps in GLR.

("Best practices for microservices" **AND** "Performance") OR ("Microservices deployment best practices" **AND** "Performance") OR ("Microservices orchestration best practices" **AND** "Performance")

4.1.1. Information sources and search queries

We identified two main sources of documents for our GLR: *Google Search Engine* and *Google Scholar*. We formulated the following search query: The search query is formulated to specifically contain the terms "performance", "microservices", and "best practices".

4.1.2. Inclusion and exclusion criteria

We executed the search query separately in *Google Search Engine* and in *Google Scholar Engine*, and we retrieved 88 and 14 documents, respectively, for a total of 102 documents. We manually removed duplication and we retained 75 documents. We the applied the following inclusion and exclusion criteria to select the documents to be analyzed.

- **Inclusion Criteria:**
  - The document must be a white paper, blog post, industrial technical report, or any non-peer-reviewed document.
  - The document reports techniques, principles, or architectures principal related to optimising performance in microservices.
  - The document provides empirical evidence of performance improvements from applying SE practices.
  - The document demonstrates a practical, real-world implementation of best practices.
- **Exclusion Criteria:**
  - The document is not in English.
  - [R2C1] The document is outside the scope of microservices performance and deployment practices after title/abstract and, when needed, full-text screening.
  - The document focuses entirely on theoretical concepts and lacks concrete examples or details.
  - The document is not authored by experts by verifying the profile of the author.

[R2C1] To avoid missing relevant grey literature, we did not restrict our GLR search to documents whose titles explicitly mention "best practices" or "microservices." Instead, all retrieved sources were screened in a multi-stage process. We first examined titles and abstracts to assess topical relevance. When relevance could not be determined from these fields alone, we inspected the full text. Documents were retained or excluded strictly based on the inclusion and exclusion criteria defined in Section 4.1.2, rather than on the presence or absence of specific keywords in the title. This approach increases recall while ensuring that only sources addressing microservices-based performance and deployment practices were included.

The first author of this paper applied the first two exclusion criteria and excluded 24 documents to retain 51 documents. The first two authors of this paper applied the last 3 exclusion criteria and removed 30 documents, resolving disagreements through discussions involving all the authors of this paper. We finally selected 20 documents at this step.

4.1.3. [R2C1] quality and risk-of-bias assessment

Because GLR sources are non-peer reviewed, we conducted a lightweight quality and risk-of-bias appraisal. For each document, we recorded (i) the type of source (e.g., vendor white paper, engineering blog, personal blog), (ii) whether the authorship and organizational affiliation were clearly stated, and (iii) whether the document provided concrete implementation details and empirical

Table 3
Exhaustive list of SE practices.

#	Practices Title	References
1	Maintain clear code and versioning	[54–59]
2	Use an API gateway	[34,56,60–64]
3	Use efficient communication protocols and communication Patterns	[61,65,66]
4	Use request management (e.g., optimize payload, load balancing)	[67–69]
5	Utilize specialized databases like NoSQL or in-memory databases	[70–72]
6	Use a database per microservices to maintain data isolation	[60,62,70–73]
7	Implement health checks and Utilize monitoring tools	[74–76]
8	Use containerization for microservices for lightweight, consistent, and portable deployment	[26,56,69,77–79]
9	Use continuous integration (CI)	[57,59,79,80]
10	Use continuous deployment (CD) pipelines and automation pipelines	[26,57,59,69,79–83]
11	Monitor microservices performance in real-time to identify and address potential bottlenecks	[75,76,84]
12	Use bundling and Minification	[85,86]
13	Use deployment practice like canary release pattern, blue-green deployment	[57,59,87,88]

performance evidence. Based on these indicators, we classified each source as low, moderate, or high risk of bias. Practices supported only by high-risk sources were either discarded or treated as tentative and not included in our final catalog. Our final set of practices is therefore primarily supported by low- and moderate-risk sources. Most documents (70%) came from vendor or large-scale engineering blogs with detailed case descriptions and were rated low or moderate risk; only a minority were rated high risk and did not affect the final set of practices.

4.1.4. Snowballing

To broaden the scope of our literature review, we performed a snowballing technique where we examined the references and citations within our initial set of documents. As a result, we included 15 additional relevant documents, after applying the same inclusion/exclusion criteria, for review.

Overall, we included 35 documents in our review. These documents are publicly accessible in the replication package available on Zenodo or Ptidej website. Furthermore, the industrial studies that we referenced in this study can be accessed online by consulting the links provided in the replication package.

4.1.5. Compilation of results

We collected any practices or recommendations from the various documents found into a single list. We selected the most recommended SE practices to formulate our catalog of practices in response to **RQ2**.

4.2. Analysis

We focused on the performance optimization and resource utilization challenges of the microservice-based IoT systems on the edge. We assessed whether the document reported how the selected practices can help in addressing those challenges. We identified and selected the most suitable practices that align with the core focus of our study. For example, practices such as service granularity, asynchronous communication, and load balancing are directly related to optimizing performance in resource-constrained edge environments. By linking the identified practices with microservice-based IoT systems challenges on the Edge, we narrowed down the list of practices to 13 less relevant in this context. Table 3 summarizes the practices we identified.

We identified some SE practices for each of the following challenges identified in SLR:

Placement Problem. Based on the findings in [78,79], *containerized microservices* are frequently covered in the literature, because they provide a way for isolating systems. Containers provide fundamental advantages, by including minimum resource consumption and providing the ability to execute each container with its own separate file system and resources while sharing the host operating system. Orchestration platforms like Kubernetes facilitate container portability and automated management via capabilities like health monitoring, resource allocation, and configuration control [56]. Orchestration platforms together with containers standardize environments and automate scaling and administration [69,81–83], enhancing the efficiency and reliability of microservices deployments. Continuous deployment is another relevant SE practice that aims to automate and streamline the process of deploying the changes made on system code into production environments. This practice integrates deployment-ready pipelines and tools like Infrastructure as Code (IaC). IaC automatically helps to configure environments and build infrastructure and allows environments to be provisioned in a code-driven, reusable manner. This practice allows developers to address performance challenges [26,56,77].

Another relevant practice involves *minification and bundling techniques* [85,86], which are designed to enhance the preparation of microservices for production. These techniques aim to optimize the packaging and deployment of microservices and help to ensure that microservices can deliver high-quality service in real-world scenarios. These practices may improve the performance of microservice-based IoT systems on the Edge.

Network. Among the various design practices, the *API Gateway* practice stands out as an architectural choice discussed in the documents to address this challenge. It serves as a centralized entry point for requests and helps manage the flow of traffic between microservices. Not having a *API Gateway* can be considered as a bad practice [34,60–64]. Notably, companies like Netflix have adopted this practice, further emphasizing its significance [89]. Other SE practices like *Service Mesh*, *Circuit Breaker Pattern* [62],

Takeaway from RQ2: SE practices that may improve the performance and resource utilization of microservices-based IoT systems on the Edge include (1) API gateway for REST API exposure, (2) RPC for efficient microservices' communication, (3) database-per-service approach with specialized NoSQL or in-memory databases, (4) mono-repository versioning strategy for maintainability, and (5) containerization, (6) minification and bundling techniques to craft lightweight and portable system components.

and geographic distribution of services can help reduce latency and improve the resilience, reliability, and responsiveness of the microservices-based system. These SE practices control communication, prevent system failures, and distribute services over multiple regions to minimize latency and mitigate localised failures. Furthermore, proper *request routing*, *load balancing* [90], *loose coupling* are other SE practices considerations to mitigate these challenges.

Some documents (e.g., [17,72,91]) recommend that event driven messaging between microservices and autoscaling can improve performance. These recommendations can help to address common microservice network issues such as complexity, congestion, cascading failures, and efficiency. Additionally, implementations such as Remote Procedure Call (RPC) frameworks (e.g., gRPC) offer performance advantages due to their use of Hypertext Transfer Protocol (HTTP) (i.e., HTTP/2) for efficient, multiplexed streams and lower latency. Protocol Buffers enable compact message sizes, which speed up transmission and save bandwidth. With support for data streaming and clearly defined service contracts, RPC frameworks implementing request/reply optimize inter service communication, making them highly effective for microservices, leading to better performance [65,66,92].

Moreover, some other techniques, such as "*optimising payload size*", "*auto-scaling services*", "*connection pooling*", "*rate limiting*" [67], and "*enabling fault tolerance*" [68,69] can contribute to more efficient and reliable systems.

Programming Complexity. We identified SE practices in some documents [56–59] that focus on facilitating clean, maintainable code. Many coding practices draw inspiration from DevOps methodologies, emphasizing configuration management, version control, and proper API versioning [57,59]. Tracking all code changes through version control [58] ensures transparency and accountability. Additionally, the adoption of a "*configuration management strategy*" known as the "*mono-repo versioning strategy*", as exemplified by Google [55], has gained prominence. This SE practice involves consolidating code repositories into a single, repository, streamlining code management and version control. Furthermore, insights from industry leaders like Netflix [56] underscore the importance of code consistency within a microservices' architecture. When it becomes necessary to introduce new code or make modifications within successfully deployed microservices, a common practice is to create new microservices tailored specifically for the new functionality. This SE practice preserves the stability and reliability of existing microservices and facilitates the evolution of new features and functionalities.

Resource Optimization. We identified various SE practices for *data storage, retrieval, synchronization*, as well as ensuring *data consistency and resilience* in distributed systems. Adding caching layers can help in reducing duplicate requests to databases or external services, to avoid repeating resource-intensive operations. One of SE practices, as highlighted by the authors in [56,93], is the "*Database Per Service Architectural Practice*", which enables each service to function, expand, and deploy independently, thereby avoiding tight coupling with other services. Furthermore, authors in [60] viewed this practice as a solution to address the anti-pattern known as "*Shared Persistence*", which maintains the separation of concerns and optimizing modularity. Another relevant pattern is the "*Command Query Responsibility Segregation (CQRS)*" SE practice, which is advised for systems experiencing high volumes of traffic [70,71], and which allows for the separation of read and write operations, which can lead to more optimized and efficient handling of data operations. Database performance also benefits from meticulous tuning and indexing [72], ensuring efficient query processing and optimal resource use. Together, these practices aim to eliminate redundant or inefficient work, reduce contention, allow independent scaling, and improve performance. The focus is on making optimal use of computing resources by structuring the architecture and system to avoid waste and bottlenecks. Another key practice for resource optimization is the use of "*Minification and Bundling Techniques*" [85,86], which can optimise the build process by minifying code and bundling together resources, resulting in smaller artifact sizes. This reduces the amount of data that needs to be transferred and loaded by the system, thereby optimising bandwidth utilisation, memory usage, and load times. Improving resource delivery, minification, and bundling leads to more efficient resource usage. Furthermore, maintaining a single container to run all components of the system is another SE practices. This SE practice, related to deployment policies, ensures the optimal balance between performance and power consumption [17].

4.3. Conclusion

We identified 6 SE practices that can improve the performance and resource utilization of microservices-based IoT systems on the Edge. While many of these practices are used in production, they have not been systematically or empirically evaluated in the specific context of Edge based IoT deployment. The next phase of our study involves conducting an empirical study to assess the performance and resource utilization impact of these practices.

Table 4
SE practices evaluated.

#	SE Practice (GLR)	Ev.	Explicit Hypothesis
1	API gateway for REST API exposure	Yes	Using an API gateway improves request routing efficiency and reduces latency compared to direct REST exposure.
2	RPC (e.g., gRPC) for microservices communication	Yes	gRPC reduces communication latency compared to REST under equivalent workloads.
3	Database-per-service with specialized NoSQL / in-memory DBs	Yes	A database-per-service approach improves query performance and scalability compared to a shared relational database.
4	Mono-repository versioning strategy	No	Not evaluated, as its impact is primarily on maintainability rather than runtime performance or resource utilization.
5	Containerization	Yes	Containerized deployments provide better resource isolation and portability with minimal overhead compared to native processes.
6	Minification and bundling techniques	Yes	Minification and bundling reduce transfer times and resource consumption on constrained edge devices compared to unoptimized components.

4.4. Synthesis of SLR and GLR

Our Synthesis shows that microservices adoption and containerization are consistently reported across both academic and industry sources (7 SLR articles; 15 GLR sources). Other practices, such as API gateways, per service databases, and bundling and minification, appear mainly in industry sources, highlighting their pragmatic and fast evolving nature.

4.5. Implication for developer effort

In terms of developer effort, microservices typically require more orchestration and configuration (for example, communication between services, deployment pipelines, and monitoring), which increases management complexity. By contrast, monolithic systems are simpler to configure and maintain as a single unit, although they offer less flexibility for scaling or modular changes. We explicitly leave a systematic evaluation of sustained CPU-pressure responsiveness, degradation thresholds, and developer effort to future work.

5. Empirical study

This section addresses **RQ3: What is the performance impact of some of the identified SE practices for microservices-based IoT systems deployed on the Edge**? We conducted an empirical study to validate some of the identified SE practices. Table 4 shows the practices we evaluated in this study.

We used WIMP as case study, and we carried out a comparative experiment involving two versions of WIMP, one without and another with the selected SE practices. WIMP[3] or "Where Is My Professor", is an IoT-based system that enables students to track the availability of their professors in real-time. The main objective of this system is to collect data from various IoT devices, such as sensors, cameras, and beacons, to provide an automated response on the professor's availability by analysing the collected data. This system has smart sensors, such as the WEMO smart plug[4], as well as devices like the Fitbit Watch[5], which serve the purpose of location tracking and play a pivotal role in the decision-making process. WIMP is an open source proof of concept IoT platform used as our subject system, available in both monolithic and microservices deployments. We selected the following most recommended SE practices: API gateway, containerization, database per service, minification and bundling technique, and RPC.

In the next sections, we provide an overview of our experiment design, use case, metrics used in comparison, test scenarios, and analysis. For the remainder of this section, we will use "microservice-based IoT system" to refer to the WIMP version based on the selected SE practices, while "legacy system" refers to the WIMP version without the selected SE practices.

5.1. Experiment design

We adopted a "one-factor, two-level" experimental design [94]. We conducted an experiment to investigate the effect of a single variable-the "use of SE practices" as a bundled factor-across two settings (the two versions of WIMP): *absent* (legacy/monolithic) vs.

[3] https://ptidejteam.github.io/wimp-wiki
[4] https://www.belkin.com/products/wemo-smart-home/
[5] https://www.fitbit.com/global/en-ca/home

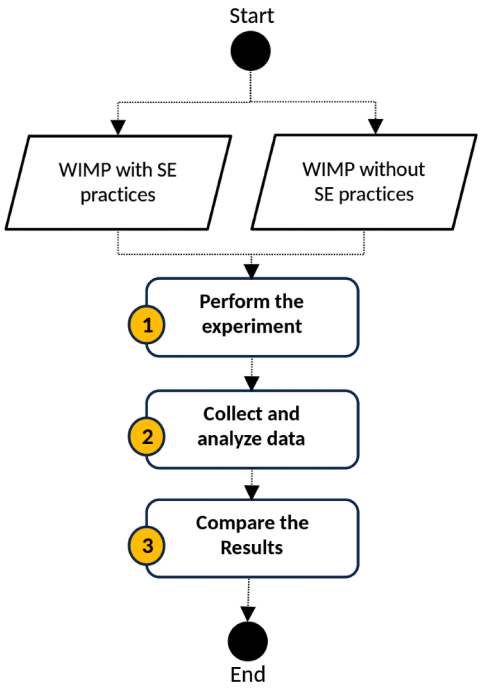


Fig. 4. Experiment design.

present (microservices). We combined the selected SE practices (e.g., API-gateway mediation, RPC-based inter-service communication, per-service database ownership, independent deployment/scaling, and service-level observability) and evaluated their collective impact on performance optimisation and resource utilisation, while holding constant the authentication/authorisation flow, core business logic, and deployment environment. This one-factor design estimates the aggregate effect of co-adopting these practices; it does not attribute effects to individual practices or their interactions, which we leave to future ablation or fractional-factorial studies. [R2C2] A full ablation or fractional-factorial design falls outside the scope of this study due to its markedly different objectives and experimental footprint; we therefore treat this as future work. [R2C4] Similarly, we intentionally did not simulate degraded-network conditions, as our design focuses on isolating architectural performance effects rather than resilience under variable network quality. Fig. 4 illustrates the process to assess and compare the performance and resource utilization of two versions of WIMP: one without these practices and another with these practices.

We migrated the existing WIMP system from a monolithic-based architecture to a microservices-based one. We created microservices based on the main functionalities of the existing WIMP, and we applied the selected SE practices. As a result, we ended up with two versions of the WIMP system: one with the applied SE practices (i.e., microservice-based one) and the existing one (i.e., monolithic-based one). **One minification and bundling,** our microservices are implemented in JavaScript (Node.js). For deployment, we use Webpack (production mode with tree-shaking and minification) to bundle each service's entry point (`index.js`) together with its dependent modules into a single optimized JavaScript artifact. Across services, the bundled artifacts are approximately 20–25% smaller than the unoptimized source trees. We also observed slightly reduced cold-start times on the Raspberry Pi, attributable to fewer dynamic module loads from the file system. The primary effect is a smaller payload to transfer and lower startup I/O on the device; functional behavior is unchanged. We modified the source code for both versions to integrate the *Prometheus client* to expose and collect various metrics. The experiment consists of the following 3 steps:

❶ **Perform the experiment**. We used load testing tool to evaluate how both WIMP versions behave under different load conditions and observe the performance and resource utilization of each. We used an open-source load testing tool called *Artillery*[6]. This tool is used to send concurrent requests and evaluate the performance of the system.

❷ **Collect and analyze data**. We collected the data for different metrics of both versions of WIMP in two ways. We collected the metrics generated by *Artillery*. This tool generates detailed reports that include latency, throughput, and number requests.

We used *Prometheus*[7] and *Grafana*[8], to collect and visualize CPU and memory usage.

❸ **Compare the results**. We conducted a comparative analysis of the data from those two systems. This analysis allowed us to evaluate the performance of each system.

[6] https://www.artillery.io/docs
[7] https://prometheus.io/
[8] https://grafana.com/

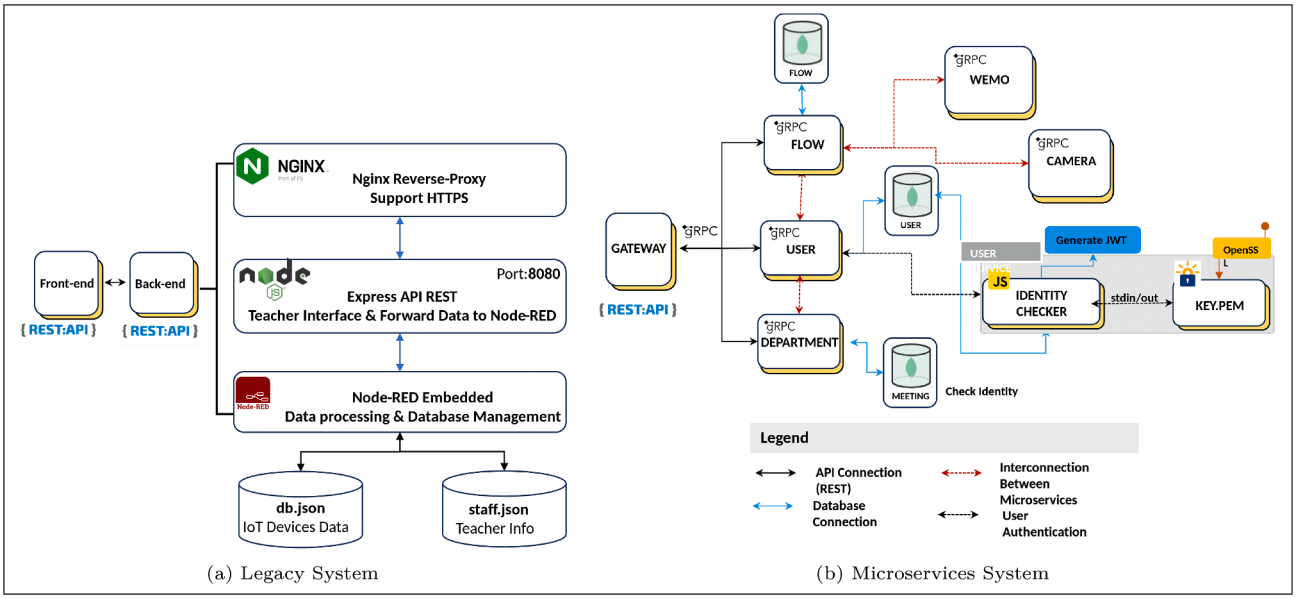


Fig. 5. Architectural design of the two WIMP implementations.

5.2. WIMP description

WIMP[9] is an IoT-based system that enables students to track the availability of their professors in real-time. This system collects data from various IoT devices, such as sensors, cameras, and smartwatches, to learn where professors are located inside the university building and other pertinent information. The system can offer an automated response on the professor's availability by analyzing the collected data.

The intended scenario attempts to improve the planning and communication between students and professors. With this system, students can use sensors to determine whether the professor is in their office, engaged in conversations with others, and receptive to interaction with students. This system has smart sensors, such as the WEMO smart plug[10], as well as devices like the Fitbit Watch[11], which serve the purpose of location tracking and play a pivotal role in the decision-making process. The main functionalities will be summarised as follows:

1. Internal users' management: this includes user accounts, profiles, permissions, etc.
2. Automated responses about professor's availability generated through analysis of the collected data from various IoT devices.
3. Real-time tracking of professor's availability and location using data collected from IoT devices.

The ultimate goal of developing WIMP is to provide a functional IoT system to researchers to conduct experiments on testing.

5.2.1. Legacy IoT system

At the architectural level, WIMP comprises two main parts: the student part and the professor/admin part. The first part of the WIMP system gives students and researchers access to a web interface for tracking their professors, which is referred to "front-end of the WIMP system". The second part represents the core part of the WIMP project is "back-end of the WIMP system". It has GUI that allows professors to define and manage their status and construct their logic in a Node-RED flow to compute their current state. For reference, the system repositories are publicly available on GitHub: back-end repository[12], and front-end repository[13].

5.2.2. Microservices-based IoT system

We conducted an architectural redesign of the existing system and introduced microservices components. To determine which aspects of the architecture to encapsulate as microservices, we used the method in [95] to structure our microservices around the core business functionalities in Section 5.2. This transformation resulted in a microservices-based IoT system leveraging the use of API gateway. The system is decomposed into business logic components that use RPC and HTTP/2 as the main internal communication

[9] https://ptidejteam.github.io/wimp-wiki
[10] https://www.belkin.com/products/wemo-smart-home/
[11] https://www.fitbit.com/global/en-ca/home
[12] https://github.com/ptidejteam/wimp-backend
[13] https://github.com/ptidejteam/wimp-frontend

Table 5
Payload characteristics.

Scenario	Request Payload	Response Payload
Auth only	~500 B	~2 KB (Constant JSON)
Auth + Professor Availability	up to ~5 KB	~2 KB (Constant JSON)

protocol, and we made the system accessible to the outside world using HTTP/1.x as the core of almost all web applications and provides a text-based message format, in contrast to the binary format used by HTTP/2. We present an illustration of this architecture in Fig. 5.

Within our architecture, a microservice, known as the "gateway" serves as the single point of entry to our system, managing the routing of API requests to all the services, and functioning as a proxy server to address HTTP/1.x requests. Additionally, it manages functions such as authorization verification. This microservice has been implemented using *GoLang* because it provides *RPC-Gateway plugin*, which facilitates the interpretation of *protobuf*[14] service definitions and generates a reverse proxy server capable of converting a *RESTful HTTP API* calls into *RPC* calls.

The other components in the microservices-based IoT systems rely on *Node.js* as the *JavaScript* run-time environment. These components follow a database-per-service SE practice, where we used specialized databases like *NoSQL* or in-memory databases practice by using *MongoDB* as the data source platforms. In addition to *Node.js*, we used *Python* to create the microservices-specific components. These components function as the intermediary layer interacting with IoT smart devices and sensors.

Baseline parity (held constant). Both variants use the same authentication and authorization flow (JWT with the same token signing algorithm, RS256). In the microservices setup, the Go gateway terminates TLS at the edge and bridges HTTP/1.x client traffic to HTTP/2/gRPC for internal calls; services revalidate scopes consistently. Both variants use MongoDB with mirrored schemas and indexes (including TTLs and validation rules). This parity ensures that any differences stem from architectural decomposition rather than configuration.

The first microservice, "User", serves as a dedicated user management system, efficiently handling Create, Read, Update, and Delete (CRUD) operations while also managing user authentication and identity verification. The system classifies users into distinct permission levels, and custom middleware enforces access permissions by applying the "token authorization practice".

The "Flow" microservice focuses on availability assessments using data from devices and connects to device-specific microservices through Node-RED flows. This microservice communicates with other microservices such as "*camera microservice*", which uses "YOLOv3" for individual presence detection in a professor's office, and "*WEMO microservice*" to communicate with "smart-plug".

The "Department" microservice specializes in delivering comprehensive user information, accounting for user affiliations with specific departments, and supporting CRUD operations for effective user data management. In the end, as part of our system deployment within a Docker environment, we employ minification and bundling techniques to generate the built microservices.

5.2.3. Payload characteristics

In our evaluation, we considered two representative scenarios:

- **Scenario 1 (Authentication only).** The client sends an authentication request (approximately ~500 B payload using REST/JSON or gRPC). The server responds with a constant JSON payload of approximately ~2 KB. This scenario captures the baseline cost of handling secure login.
- **Scenario 2 (Authentication and professor availability).** The client first performs authentication and then retrieves professor availability. The combined request size is up to approximately ~5 KB, depending on request aggregation, while the server response remains a constant JSON payload of approximately ~2 KB. This scenario stresses service to service communication by involving authentication and an additional service call.

The response payload size is constant across both scenarios. Throughput in kbit/s is directly proportional to the number of requests per second. Table 5 summarizes the payload characteristics for both scenarios.

5.3. Test scenarios

We defined two test scenarios for the evaluation. These scenarios are based on the main functionalities described in Section 5.2.

- **Scenario 1**: Sending 10,000 requests for authentication into the IoT system.
- **Scenario 2**: Sending 10,000 requests to retrieve the professor's availability.

In both scenarios, we sent concurrent requests by varying the number of virtual users generated by the load testing tool (*Artillery*). Through an iterative process of trial and error using the tool, we selected virtual user ranges (1 to 1, 1 to 10, 1 to 50, 1 to 100, and 1 to 400) to observe the system's response to diverse user requests. Based on pre-testing rounds, we determined that sending 10,000 requests would provide a suitable benchmark for comparison.

[14] https://protobuf.dev/

Table 6
Table of metrics and tools used.

Category	Tool	Metrics
CPU Performance	Prometheus	cpu_user_seconds_total
Memory Performance	Prometheus	process_resident_memory_bytes
Network Performance	Artillery	latency, throughput

[✚] **cpu_user_seconds_total** measures cumulative cpu time spent. [✚] **process_resident_memory_bytes** measures the amount of resident memory used by a given process.

Table 7
Testing environment details.

Feature	Details
Hardware	Raspberry Pi 4B
CPU / RAM	Quad-core ARM Cortex-A72 @ 1.5 GHz / 4 GB RAM
Storage	SD Card 16–32 GB, Class 10
OS	Raspberry Pi OS 64-bit, Kernel 6.x
Docker / Compose	Docker 23.x, Docker Compose 1.29.x
Container Resource Limits	CPU shares and memory limits applied consistently
Network Topology	Single host, default Docker bridge network, Gigabit Ethernet
Ambient Contention	Minimal; no other major workloads

The first scenario simulates the user authentication process within the system. The second scenario simulates the user's request for the availability status of a professor. This scenario helps not only to assess the system's performance but also to explore the complex dynamics of IoT devices and the intercommunication between microservices, especially in the case of the microservices-based IoT system.

5.4. Metrics

To evaluate the performance of each system, we identify various metrics that draw inspiration from Lira et al. [17].

CPU Performance. To evaluate CPU performance, we assessed the *CPU Usage* metric, which indicates the percentage of CPU capacity. For example, when considering the authentication scenario referred to as "Scenario 1" in Section 5.3, the CPU usage combines the measurements from both the gateway and the user microservices within the context of the microservices-based system, while in the availability scenario, the metric includes the combination of the gateway process, the flow process, and the WEMO process. In contrast, in the legacy system, the measurements originate from a single process for both scenarios.

Memory Performance. To evaluate memory performance, we assess process memory usage, which takes into consideration the aggregate amount of physical memory. Similarly to CPU performance, those measurements involve the cumulative memory usage of the same microservices in both scenarios within the microservices-based system. However, for in both scenarios, the measurements in the legacy system come from a single process.

Network Performance. We conducted network evaluation while taking into account distinct attributes: throughput, which measures the number of requests or transactions processed per second, and latency, which assesses the duration that a request remains pending before it is processed. This evaluation is generated with the load-testing tool called "*Artillery*" for both scenarios, which is configured to automatically capture some statistics on latency and throughput.

Table 6 summarizes the metrics, the utilized tools, and their associated metrics/queries.

5.5. Testing environment

We used the *GoLang Compiler, Node.js, and Python* for the development of our microservices. *MongoDB* served as our chosen database for data storage. We deployed both versions of WIMP on the same *Raspberry Pi 4 Model B* through *Docker Compose* to enable a fair comparison. Table 7 shows the details of the environment we used for experimentation.

5.6. Conducting experiments

We conducted experiments to evaluate performance and resource utilization for both versions of our WIMP. We used "Artillery" to generate load, collected CPU and memory metrics with "Prometheus", and visualized them with "Grafana". Fig. 6 summarizes results across five Virtual User levels (1, 10, 50, 100, 400). For clarity, throughput is reported both as kbit per second and as requests per second. Each experiment was executed in multiple independent runs; for the results reported here, we used three or four runs per configuration. A short warm up period preceded measurement to allow the system to stabilize, and metrics from this period were discarded. Requests were HTTP GET generated by Artillery with 500 B total size (headers and any body). Responses were a constant JSON payload of 2 KB with content type application/json. Because request and response sizes were held constant across runs,

Table 8
Experimental characteristics.

Aspect	Description
Run count	Each experiment executed 3–4 independent runs.
Warm up policy	Artillery applies a default warm up and stabilization period before measurements are recorded; no additional manual warm up was introduced.
Median latency	For Scenario-2, we report the median, p90, p95, and p99 latency across independent runs for all tested VU levels (see Table 9). These distributional metrics complement the mean and characterise tail latency under increasing load.
Confidence intervals / Effect sizes	For Scenario-2, 95% CIs of the mean latency are computed across runs; formal effect sizes are not reported due to the small run count.
Statistical tests	Not applied in this study (e.g., Mann-Whitney U or Welch t test); noted as valuable for future refinements.
Request characteristics	Artillery default HTTP GET requests, ∼500 B (headers + body).
Response characteristics	Constant JSON payload, ∼2 KB (application/json).
Throughput units	Reported in kbit/s, directly proportional to req/s since payload size was constant.

Table 9
[R2C3] latency distributional metrics for Scenario-2.

Metric	Mean (s)	Median (s)	p90 (s)	p95 (s)	p99 (s)	Min (s)	Max (s)
VU-1	0.085	0.078	0.148	0.182	0.244	0.03	0.27
VU-10	0.39	0.35	0.68	0.82	1.13	0.12	1.25
VU-50	1.88	1.72	3.12	3.76	5.05	0.52	5.48
VU-100	3.52	3.21	5.68	6.88	9.42	1.09	10.14
VU-400	12.83	11.92	18.76	21.94	29.88	5.16	32.15

throughput reported in kbit/s is directly proportional to req/s; we report both units for clarity. The characteristics of our experiment are summarized in Table 8.

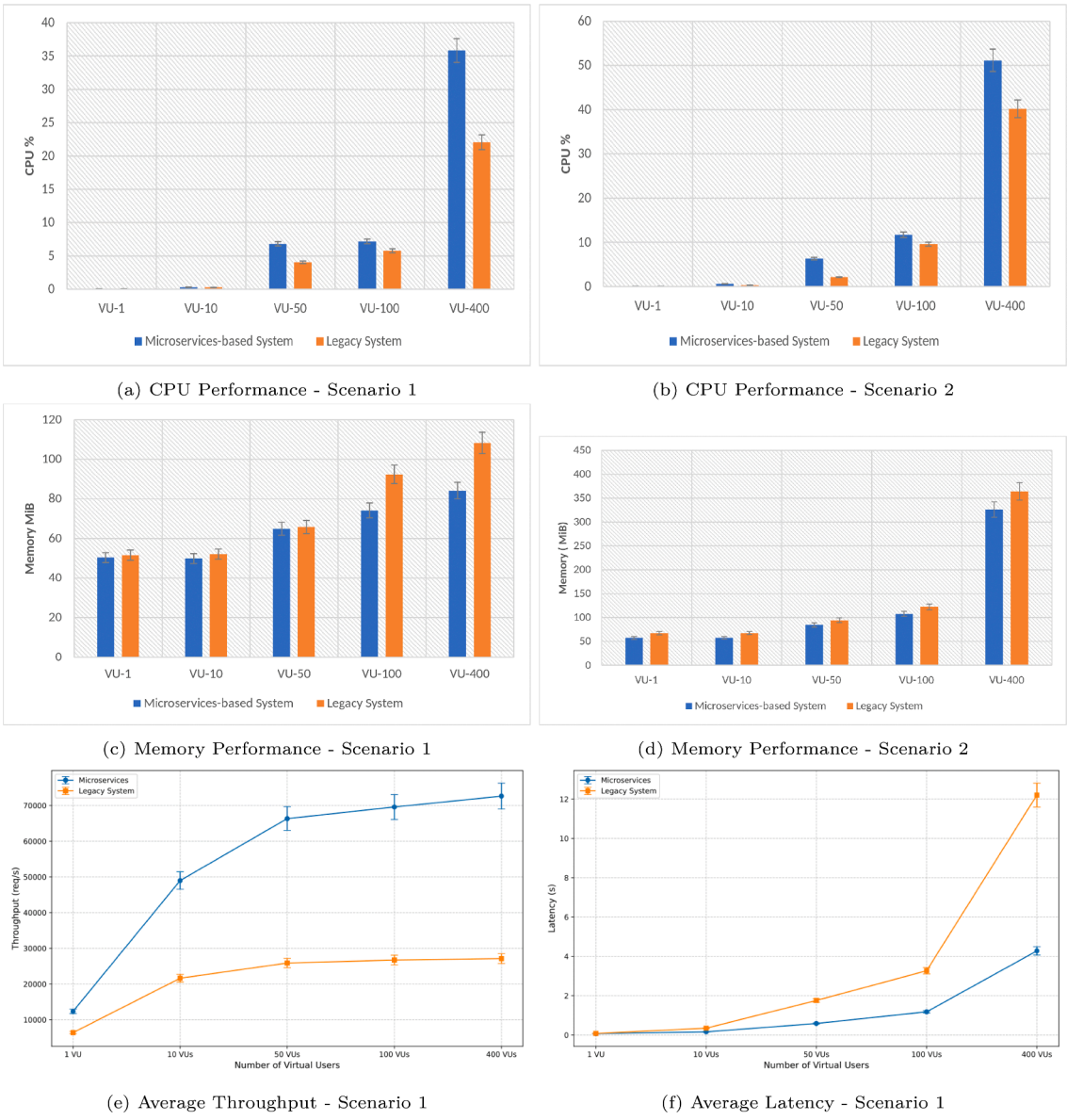
[R2C3] Table 9 reports the latency distribution metrics for Scenario-2 across different load levels. As the number of virtual users (VU) increases from 1 to 400, both central tendency measures (mean and median) and tail latencies (p90, p95, p99) increase substantially. For low load (VU-1 and VU-10), median latency remains below 0.4 s and even the p99 latency stays below 1.2 s, indicating consistently fast responses. At higher loads, especially for VU-100 and VU-400, mean latency increases to 3.52 s and 12.83 s respectively, while p95 and p99 latencies reach up to 21.94 s and 29.88 s. The growing gap between the median and the upper percentiles, along with the widening min-max range, indicates a pronounced long-tail behavior and increasing response-time variability under heavy concurrency.

5.7. Analysis

CPU Performance. Fig. 6a and b show the CPU performance under five different VU scenarios while executing both systems. The results show that the microservices-based IoT system requires higher CPU usage. This is primarily due to dissolving the legacy system, which consists of individual processes, into different microservices that collectively form the foundation of the microservices-based IoT system. In the initial scenario, the microservices-based IoT system showed a CPU usage ranging between 0.04% and 35.85% as we changed the number of VUs from 1 to 10, 50, 100, and 400. While the CPU usage for the legacy system ranges from 0.07% to 22.07%. In the second case, when the number of VU varied from 1 to 10, 50, 100, and 400, the microservices-based IoT system showed CPU utilization ranging from 0.10% to 51.15% while for legacy system ranged from 0.04% to 40.20%.

Memory Performance. We examined the data in Fig. 6c and d, and we noticed that the microservices-based system exhibits less memory compared to the legacy system. To elaborate, in the first scenario, we observed memory savings of 2.36%, 4.29%, 1.5%, 19.76%, and 22.24% as VU ranged from 1 to 10, 50, 100, and 400 when compared to the legacy system. Moreover, for the second scenario, we observed memory saving of 14.85%, 14.78%, 10.25%, 11.94%, and 10.49% within the microservices-based system compared to the legacy. We do notice some fluctuation in the percentage of memory savings in the first scenario. We attribute these variations to dynamic memory allocation and garbage collection cycles that run periodically.

Network Performance. Fig. 6e and f show the average throughput and latency of both IoT systems across various virtual user configurations during testing. Throughput is reported both as kbit/s and as requests per second (req/s) to avoid ambiguity. We present Scenario 1 in the figures and provide Scenario 2 network metrics-throughput (kbit/s and req/s) and latency (median, p95, p99) across independent runs. Both scenarios used identical payload sizes. We noticed a consistent performance advantage for the microservices based IoT system for both throughput and response time, irrespective of the number of VUs involved. We observed that the microservices based IoT system showed a throughput of 93.07%, 126.50%, 156.25%, 160.28%, and 167.81% higher than the legacy system as the number of VUs ranged from 1 to 10, 50, 100, and 400. Request and response sizes were held constant across runs (HTTP GET ∼500 B; constant JSON response ∼2 KB, application/json). Accordingly, higher throughput in kbit/s implies higher req/s for the same workload. The increased throughput is likely a contributing factor to the higher CPU utilization we observed with



(a) CPU Performance - Scenario 1

(b) CPU Performance - Scenario 2

(c) Memory Performance - Scenario 1

(d) Memory Performance - Scenario 2

(e) Average Throughput - Scenario 1

(f) Average Latency - Scenario 1

Fig. 6. Performance results for different VU scenarios. The labels on the x-axis are as follows: VU-1: 1 virtual user, VU-10: 10 virtual users, VU-50: 50 virtual users, VU-100: 100 virtual users, and VU-400: 400 virtual users.

the microservices based system, as it is processing more requests per second at peak load. Moreover, the microservices based IoT system showed a latency of 12.5%, 54.28%, 66.47%, 63.91%, and 64.94% lower compared to the legacy system when varying the number of VUs, indicating consistently higher delays for the legacy system. Unless otherwise stated, overall throughput and latency effects are computed as the arithmetic mean of the per VU percentage changes across the five VU levels (unweighted).

5.8. Trade-offs beyond performance

When transitioning from a monolithic architecture to a microservice-based approach in IoT systems, teams encounter significant performance and memory trade-offs. Decomposing a large system into smaller, manageable services can enhance flexibility and ease updates, allowing each component to scale independently. However, this fragmentation requires careful coordination, especially when services need to share data, which can introduce latency and complicate memory management. While tools like API gateways and containers streamline setups and accelerate onboarding for new team members, they also add layers of complexity, increasing the potential for failures. Consequently, the shift in workflow moves from managing a single system to orchestrating numerous interconnected services, necessitating vigilant monitoring of deployments and performance metrics to ensure optimal functionality.

Takeaway from RQ3: The microservices based IoT system consumes more CPU than the legacy system; CPU load increases by 1.04% in Scenario 1 and by 1.14% in Scenario 2(averaged across VU levels). We noticed a reduction in memory utilisation, leading to average savings of 6.98% in Scenario 1 and 12.95% in Scenario 2 compared to the legacy system. Averaged (unweighted) across the five VU levels in Scenario 1, the microservices based IoT system showed a 134.02% improvement in throughput and 49.29% lower latency compared to the monolithic based IoT system. We found the CPU usage to be an acceptable trade off considering the improvements in latency and throughput.

5.9. General observations

This section documents the practices absent in the legacy version and the elements held constant across both versions.

1. **Practices absent in the legacy version:**
   - Independent service deployment and scaling (the monolith is deployed as a single unit).
   - Service-level observability (fine-grained logs, metrics, and tracing are only available in the microservices version).
   - API gateway mediation (not present in the monolith; client requests are handled directly).
   - Per-service databases and clear schema ownership (the monolith uses a shared database).
2. **Practices held constant in both versions:**
   - Authentication and token-based authorization flow.
   - Core business logic and feature set.
   - Deployment environment (same host hardware, operating system, container runtime, and orchestration with Docker Compose).

5.10. Conclusion

Our empirical evaluation provides valuable insights into the performance trade-offs of using SE practices in a microservices-based IoT system compared to a legacy IoT system. Although we observed increased CPU usage with the microservices' architecture, implementing SE practices led to notable gains in memory efficiency, throughput, and latency compared to the legacy system.

6. Threats to validity

There are threats to the validity of this study in terms of construct validity, internal validity, and external validity.

Construct Validity. Construct validity pertains to the selection of papers analyzed and the methodology for extracting data from those papers. One potential concern is the omission of relevant papers. To address this issue, we selected papers from the top five digital libraries, a common practice in most software engineering literature reviews [96]. Another potential concern is bias in the paper selection process. To mitigate this risk, two of the authors conducted paper selection independently, adhering to predefined inclusion/exclusion criteria, and subsequently compared their results. Any discrepancies in the results were resolved through discussion, leading to a consensus between the authors. Yet another potential concern involves bias in data extraction. Since data extraction is a manual process, it is susceptible to personal bias and errors. To address this concern, the two authors engaged in discussions to resolve any inconsistencies. Another limitation of our work is the reliance on "Gray literature" comprised of non-peer-reviewed materials such as blogs, technical reports, and white papers. While increasingly used in software engineering research, GLR carries inherent risks due to potentially lacking scientific rigor or detailed analysis [97]. Our study aims to identify best practices and evaluate performance based on GLR, the inconsistent quality of these documents threatens the validity of our results [98]. The choice of practices, metrics, and the case study for our experiment could pose additional threats. We opted for WIMP because we did not find any free open-source IoT system publicly available. Since we already had a monolithic version of WIMP, we created a microservices-based version of this system. The selection of practices and metrics was made by the authors, but we plan to experiment with other practices and metrics in our future work. Lastly, Our study measures runtime performance only. We did not empirically assess maintainability, onboarding time, or developer cognitive load; we discuss likely implications qualitatively and plan to evaluate them in future work. Scenario 2 may suffer from gateway saturation that overshadows downstream effects, which limits the interpretability of its network metrics.

Internal Validity. Internal validity concerns the methods used in this paper. One of the possible threats pertains to *the completeness of the review*. This study focused on challenges and deployment practices for microservice-based IoT systems. However, we may have missed some challenges and practices if they were not published. However, we believe that this threat can be acceptable since we considered white papers, technical reports, and blog posts in addition to peer-reviewed papers. Another threat to the validity is the use of one system to evaluate the effectiveness of our deployment practice. Given the challenges of finding more systems to use in our experiments, we plan to conduct further experiments with other systems if available. Furthermore, future experiments could assess performance in two modes: a "burst mode" where requests are injected as fast as possible to stress test the system, and a "normal mode" with requests at regular intervals to simulate normal usage conditions. Comparing results from these two would provide deeper insight into system performance. Another potential threat concerns the data collection and analysis by the authors. To minimize author bias, we utilized load testing and monitoring tools for data collection and analysis. Another potential concern is the use of a simulated testing environment, which may not fully replicate all aspects of the system's behaviour in a real production environment

and could potentially impact performance results. Nevertheless, we consider this acceptable since real-world scenarios were not available for our study. We queried four databases with title-only fields to increase precision. This choice can reduce recall and may exclude studies that mention key terms only in their abstracts/keywords. We mitigated this risk by conducting backward and forward snowballing and a sensitivity check using synonym-expanded queries; neither yielded additional studies. Some relevant studies may still have been missed; this is a common limitation in systematic literature reviews. [R2C2] On ablation and factorial analysis, we recognize that isolating the effect of individual SE practices (e.g., RPC, database-per-service, minification/bundling) would require an ablation or fractional-factorial design. However, such designs were out of scope for this study, whose objective was to evaluate the collective impact of adopting a coherent set of practices recommended in industrial microservice literature. We therefore report the bundled effect and explicitly treat fine-grained attribution as a direction for future work, where a dedicated experimental setup can be developed. [R2C1] Lastly, in the initial design, our GLR search relied heavily on title keywords, which can reduce recall and exclude documents that mention key concepts only in the abstract or body. We re-ran the GLR search without enforcing title-level constraints and screened titles, abstracts, and, when needed, full texts against our inclusion/exclusion criteria. We also conducted a simple quality and risk-of-bias appraisal of GLR sources based on source type, authorship clarity, and presence of concrete implementation and performance evidence. Although some bias may remain, GLR inherently favors visible and well-documented practices and we believe these additional steps reduce the risk that our catalog is driven by a few low-quality or unrepresentative sources. Nonetheless, we cannot completely rule out selection bias in the GLR; some relevant practices may still be absent from public sources or described only in internal reports.

External validity. External validity refers to generalizing our study to all microservice-based IoT systems testing. Our study focused on IoT specific studies, which led us to exclude general and well cited microservices studies. This is a deliberate delimitation rather than an oversight because our focus is limited to the adoption of microservices in IoT systems. Our choice to focus on developing our use case was driven by the lack of applications that operate effectively in both legacy and microservices modes. In the future, we want to explore additional use cases and investigate the individual impact of SE practices. Furthermore, our evaluation relies on a single open-source testbed (WIMP). While this enables a controlled comparison, using one system limits generalizability. Findings may differ when replicated on multiple, larger, and more complex IoT systems. [R2C4] Furthermore, our workload was executed over a stable network; we did not induce degraded edge/fog conditions (e.g., increased RTT, packet loss, jitter, or intermittent connectivity). We intentionally excluded such conditions because the goal of RQ3 is to isolate the architectural impact of adopting SE practices under controlled load, not to evaluate resilience or QoS degradation. Simulating impaired networks would require a different experimental setup aimed at fault tolerance and robustness analysis, which is outside the scope of the current study. We therefore treat degraded-network scenarios as future work and plan to incorporate controlled network emulation in subsequent replications to strengthen generalizability.

7. Conclusion and future work

We identified the current challenges related to microservices-based IoT systems deployment on the edge, with resource utilization and performance optimization being the most reported challenges. We compiled a catalog of software engineering practices that can address these challenges and found that containerized microservices, the use of an API gateway, and employing a database per microservice are the most prevalent SE practices. We conducted an empirical study to implement these practices and compared their impact. We used throughput and latency metrics for performance evaluation, along with CPU and memory usage metrics to assess resource utilization. The results demonstrated significant improvements in throughput, latency, and memory savings when applying these practices. However, we observed that an increase in system complexity may lead to higher total CPU usage. Our future work will investigate the individual impacts of these practices and assess their effects on larger, more complex IoT systems. There is also an opportunity to expand the evaluation to include additional practices and scenarios. We will further explore potential combinations of these practices to provide concrete implementation guidelines for practitioners. We will synthesize general microservices evidence and evaluate its transferability to IoT/Edge deployments. Future work should also consider comparing event-driven, pub/sub, and CQRS with traditional RPC under edge constraints; evaluating maintainability, evolvability, and fault isolation using repo/ops metrics (lead time, change-failure rate, API churn, MTTR, blast radius) and small maintenance tasks; and gathering practitioner feedback through interviews, surveys, and an industry case study. Lastly, We explicitly leave a systematic evaluation of sustained CPU-pressure responsiveness, degradation thresholds, and developer effort to future work.

8. Replication package

The replication packages can be accessed online.

1. On the Ptidej website: https://www.ptidej.net/downloads/replications/iotj25a/
2. On the Zenodo website: https://zenodo.org/records/15127718

CRediT authorship contribution statement

Yahia El Fellah: Writing – original draft, Software, Resources, Data curation, Conceptualization; **Jean Baptiste Minani:** Writing – review & editing, Writing – original draft, Software, Resources, Methodology, Investigation, Formal analysis, Data curation, Conceptualization; **Naouel Moha:** Writing – review & editing, Validation, Supervision; **Julien Gascon-Samson:** Writing – review

& editing, Writing – original draft, Validation, Supervision; **Yann-Gaël Guéhéneuc:** Writing – review & editing, Writing – original draft, Supervision.

Data availability

Data will be made available on request.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] E. Al-Masri, Enhancing the microservices architecture for the Internet of Things, in: 2018 IEEE International Conference on Big Data (Big Data), IEEE, 2018, pp. 5119–5125.
[2] N. Khezemi, J.B. Minani, F. Sabir, N. Moha, Y.-G. Guéhéneuc, G. El Boussaidi, A systematic literature review of IoT system architectural styles and their quality requirements, IEEE Internet Thing. J. 11 (23) (2024) 37599–37616.
[3] Z. Chenail-Larcher, J.B. Minani, N. Moha, Test generation from use case specifications for IoT systems: custom, LLM-based, and hybrid approaches, in: Proceedings of the 18th IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2025.
[4] J.B. Minani, Y.E. Fellah, F. Sabir, N. Moha, Y.-G. Gu'eh'eneuc, M. Kuradusenge, T. Masuda, IoT Systems testing: taxonomy, empirical findings, and recommendations, J. Syst. Softw. (2025) 112408.
[5] J.B. Minani, F. Sabir, N. Moha, Y.-G. Guéhéneuc, T. Masuda, TISSEA: A framework for testing IoT systems based on technical software engineering aspects, IEEE Internet Thing. J. (2025).
[6] J.B. Minani, F. Sabir, Y.E. Fellah, N. Moha, Practical guidance for iot systems testing: a taxonomy, in: Proceedings of the ACM/IEEE 6th International Workshop on Software Engineering Research & Practices for the Internet of Things, 2024, pp. 57–64.
[7] J.B. Minani, Y.E. Fellah, S. Ahmed, F. Sabir, N. Moha, Y.-G. Gu'eh'eneuc, An exploratory study on code quality, testing, data accuracy, and practical use cases of IoT wearables, in: 2024 7th Conference on Cloud and Internet of Things (CIoT), IEEE, 2024, pp. 1–5.
[8] J.B. Minani, F. Sabir, N. Moha, Y.-G. Gu'eh'eneuc, A multimethod study of Internet of Things systems testing in industry, IEEE Internet Thing. J. 11 (1) (2024) 1662–1684.
[9] J.B. Minani, F. Sabir, N. Moha, Y.-G. Gu'eh'eneuc, A systematic review of IoT systems testing: objectives, approaches, tools, and challenges, IEEE Trans. Softw. Eng. 50 (4) (2024) 785–815.
[10] J.B. Minani, F. Sabir, Y.E. Fellah, N. Moha, Towards an automated approach for testing iot devices, in: Proceedings of the ACM/IEEE 6th International Workshop on Software Engineering Research & Practices for the Internet of Things, 2024, pp. 22–29.
[11] W. Shi, J. Cao, Q. Zhang, Y. Li, L. Xu, Edge computing: vision and challenges, IEEE Internet Thing. J. 3 (5) (2016) 637–646.
[12] R. Ouyang, J. Wang, H. Xu, S. Chen, X. Xiong, A. Tolba, X. Zhang, A microservice and serverless architecture for secure iot system, Sensors 23 (10) (2023).
[13] M. S"oylemez, B. Tekinerdogan, A.K. Tarhan, Challenges and solution directions of microservice architectures: a systematic literature review, Appl. Sci. 12 (11) (2022).
[14] I. Trabelsi, B. Mahmoudi, J.B. Minani, N. Moha, Y.-G. Guéhéneuc, A systematic literature review of machine learning approaches for migrating monolithic systems to microservices, IEEE Trans. Softw. Eng. (2025).
[15] H. Siddiqui, F. Khendek, M. Toeroe, Microservices based architectures for IoT systems - state-of-the-art review, IoT 23 (2023) 100854. https://www.sciencedirect.com/science/article/pii/S2542660523001774. https://doi.org/https://doi.org/10.1016/j.iot.2023.100854
[16] C. Rath, A. Mandal, A. Sarkar, Microservice based scalable IoT architecture for device interoperability, Comput. Stand.' I ' Interface. 84 (2022) 103697. https://doi.org/10.1016/j.csi.2022.103697
[17] C. Lira, E. Batista, F.C. Delicato, C. Prazeres, Architecture for IoT applications based on reactive microservices: a performance evaluation, Fut. Gener. Comput. Syst. 145 (2023) 223–238.
[18] M. Fowler, J. Lewis, Microservices guide, Retrieved May 15 (2019) 2020.
[19] N. Dragoni, I. Lanese, S.T. Larsen, M. Mazzara, R. Mustafin, L. Safina, Microservices: how to make your application scale, in: Perspectives of System Informatics: 11th International Andrei P. Ershov Informatics Conference, PSI 2017, Moscow, Russia, June 27–29, 2017, Revised Selected Papers 11, Springer, 2018, pp. 95–104.
[20] N. Dragoni, S. Giallorenzo, A.L. Lafuente, M. Mazzara, R. Mustafin, L. Safina, Microservices: yesterday, today, and tomorrow, Presen. Ulter. Softw. Eng. (2017) 195–216.
[21] D. Heaton, J.C. Carver, Claims about the use of software engineering practices in science: a systematic literature review, Inf. Softw. Technol. 67 (2015) 207–219.
[22] IEEE Computer Society, Guide to the Software Engineering Body of Knowledge (SWEBOK V3), 2014, (Available: https://www.computer.org/education/bodies-of-knowledge/software-engineering). Accessed: 2024-03-10.
[23] C. Guidi, I. Lanese, M. Mazzara, F. Montesi, Microservices: a language-based approach, Presen. Ulter. Softw. Eng. (2017) 217–225.
[24] D. Shadija, M. Rezai, R. Hill, Towards an understanding of microservices, 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP) (2017).
[25] A. Tosatto, P. Ruiu, A. Attanasio, Container-based orchestration in cloud: state of the art and challenges, in: 2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems, IEEE, 2015, pp. 70–75.
[26] B. Butzin, F. Golatowski, D. Timmermann, Microservices approach for the internet of things, in: 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), 2016, pp. 1–6. https://doi.org/10.1109/ETFA.2016.7733707
[27] A. Balalaie, A. Heydarnoori, P. Jamshidi, Microservices architecture enables devops: migration to a cloud-native architecture, IEEE Softw. 33 (3) (2016) 42–52.
[28] D. Taibi, V. Lenarduzzi, C. Pahl, Processes, motivations, and issues for migrating to microservices architectures: an empirical investigation, IEEE Cloud Comput. 4 (5) (2017) 22–32. https://doi.org/10.1109/MCC.2017.4250931
[29] A. Razzaq, A systematic review on software architectures for IoT systems and future direction to the adoption of microservices architecture, SN Comput. Sci. 1 (6) (2020) 350. https://doi.org/10.1007/s42979-020-00359-w. https://doi.org/10.1007/s42979-020-00359-w
[30] G. Campeanu, A mapping study on microservice architectures of Internet of Things and cloud computing solutions, in: 2018 7th Mediterranean Conference on Embedded Computing (MECO), IEEE, 2018, pp. 1–4. https://doi.org/10.1109/MECO.2018.8406008
[31] T. Aksakalli, Karabey, B. Can, Ahmet Burak, Deployment and communication patterns in microservice architectures: a systematic literature review, J. Syst. Softw. 180 (2021) 111014.
[32] A.S. Gaur, J. Budakoti, C.-H. Lung, Design and performance evaluation of containerized microservices on edge gateway in mobile IoT, in: 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), 2018, pp. 138–145. https://doi.org/10.1109/Cybermatics_2018.2018.00055
[33] M. Alam, J. Rufino, J. Ferreira, S.H. Ahmed, N. Shah, Y. Chen, Orchestration of microservices for IoT using docker and edge computing, IEEE Commun. Mag. 56 (9) (2018) 118–123. https://doi.org/10.1109/MCOM.2018.1701233. https://doi.org/10.1109/MCOM.2018.1701233

[34] A. Akbulut, H.G. Perros, Performance analysis of microservice design patterns, IEEE Internet Comput. 23 (6) (2019) 19–27. https://doi.org/10.1109/MIC.2019.2951094

[35] M.J. Page, J.E. McKenzie, P.M. Bossuyt, I. Boutron, T.C. Hoffmann, C.D. Mulrow, L. Shamseer, J.M. Tetzlaff, E.A. Akl, S.E. Brennan, et al., The PRISMA 2020 statement: an updated guideline for reporting systematic reviews, Int. J. Surg. 88 (2021) 105906.

[36] A. Cooke, D. Smith, A. Booth, Beyond PICO: the SPIDER tool for qualitative evidence synthesis, Qual. Health Res. 22 (10) (2012) 1435–1443.

[37] B. Ma, H. Ni, X. Zhu, R. Zhao, A comprehensive improved salp swarm algorithm on redundant container deployment problem, IEEE Access 7 (2019) 136452–136470. https://doi.org/10.1109/ACCESS.2019.2933265

[38] F. Al-Doghman, N. Moustafa, I. Khalil, N. Sohrabi, Z. Tari, A.Y. Zomaya, AI-enabled secure microservices in edge computing: opportunities and challenges, IEEE Trans. Serv. Comput. 16 (2) (2023) 1485–1504. https://doi.org/10.1109/TSC.2022.3155447

[39] H. Li, B. Tang, W. Xu, F. Guo, X. Zhang, Application deployment in mobile edge computing environment based on microservice chain, in: 2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design (CSCWD), 2022, pp. 531–536. https://doi.org/10.1109/CSCWD54268.2022.9776307

[40] B. Tang, F. Guo, B. Cao, M. Tang, K. Li, Cost-aware deployment of microservices for IoT applications in mobile edge computing environment, IEEE Trans. Netw. Serv. Manage. (2022) 1. https://doi.org/10.1109/TNSM.2022.3232503

[41] H. Zhao, S. Deng, Z. Liu, J. Yin, S. Dustdar, Distributed redundant placement for microservice-based applications at the edge, IEEE Trans. Serv. Comput. 15 (3) (2022) 1732–1745. https://doi.org/10.1109/TSC.2020.3013600

[42] H. Sami, A. Mourad, Dynamic on-demand fog formation offering on-the-fly IoT service deployment, IEEE Trans. Netw. Serv. Manage. 17 (2) (2020) 1026–1039. https://doi.org/10.1109/TNSM.2019.2963643

[43] A. Sattari, R. Ehsani, T. Leppänen, S. Pirttikangas, J. Riekki, Edge-supported microservice-based resource discovery for mist computing, in: 2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), 2020, pp. 462–468. https://doi.org/10.1109/DASC-PICom-CBDCom-CyberSciTech49142.2020.00087

[44] S. Deng, Z. Xiang, J. Taheri, M.A. Khoshkholghi, J. Yin, A.Y. Zomaya, S. Dustdar, Optimal application deployment in resource constrained distributed edges, IEEE Trans. Mob. Comput. 20 (5) (2021) 1907–1923. https://doi.org/10.1109/TMC.2020.2970698

[45] K. Cheng, S. Zhang, C. Tu, X. Shi, Z. Yin, S. Lu, Y. Liang, Q. Gu, Proscale: proactive autoscaling for microservice with time-varying workload at the edge, IEEE Trans. Parall. Distrib. Syst. 34 (4) (2023) 1294–1312. https://doi.org/10.1109/TPDS.2023.3238429

[46] H. Zhang, Z. Liu, Y. Zhang, S. Zhan, J. Yang, G. Wang, Y. Sun, Research on deployment method of edge computing gateway based on microservice architecture, IOP Conference Series: Earth and Environmental Science 675 (1) (2021) 012164. https://doi.org/10.1088/1755-1315/675/1/012164

[47] J. Islam, T. Kumar, I. Kovacevic, E. Harjula, Resource-aware dynamic service deployment for local IoT edge computing: healthcare use case, IEEE Access 9 (2021) 115868–115884. https://doi.org/10.1109/ACCESS.2021.3102867

[48] Y. Yu, J. Liu, J. Fang, Online microservice orchestration for IoT via multiobjective deep reinforcement learning, IEEE Internet Thing. J. 9 (18) (2022) 17513–17525. https://doi.org/10.1109/JIOT.2022.3155598

[49] S. University of Oviedo Computer Science Department, Calvo Sotelo 33007. Oviedo, O. Francisco, O. Donna, Towards an easily programmable IoT framework based on microservices, J. Softw. 13 (1) (2018) 90–102. https://doi.org/10.17706/jsw.13.2.90-102

[50] C. Wu, Q. Peng, Y. Xia, Y. Jin, Z. Hu, Towards cost-effective and robust AI microservice deployment in edge computing environments, Fut. Generat. Comput. Syst. 141 (2023) 129–142. https://www.sciencedirect.com/science/article/pii/S0167739X22003314. https://doi.org/https://doi.org/10.1016/j.future.2022.10.015

[51] L. Chen, Y. Xu, Z. Lu, J. Wu, K. Gai, P.C.K. Hung, M. Qiu, IoT Microservice deployment in edge-cloud hybrid environment using RL, IEEE Internet Thing J. 8 (16) (2021) 12610–12622. https://doi.org/{10.1109/JIOT.2020.3014970}

[52] A. Gholami, K. Rao, W.-P. Hsiung, O. Po, M. Sankaradas, S. Chakradhar, ROMA: Resource orchestration for microservices-based 5G applications, in: NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium, 2022, pp. 1–9. https://doi.org/10.1109/NOMS54207.2022.9789821

[53] F. Dressler, C.F. Chiasserini, F.H.P. Fitzek, H. Karl, R.L. Cigno, A. Capone, C. Casetti, F. Malandrino, V. Mancuso, F. Klingler, G. Rizzo, V-edge: virtual edge computing as an enabler for novel microservices and cooperative computing, IEEE Netw. 36 (3) (2022) 24–31. https://doi.org/10.1109/MNET.001.2100491

[54] Anon, Google JavaScript Style Guide, n.d. Available online - 5 October 2023, https://google.github.io/styleguide/jsguide.HTML.

[55] R. Potvin, J. Levenberg, Why Google stores billions of lines of code in a single repository, Commun. ACM 59 (7) (2016) 78–87.

[56] Anon, Microservices at Netflix: Architectural Best Practices, n.d. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[57] J. Humble, D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley Signature Series, Addison-Wesley, Upper Saddle River, NJ, xxxiii, 463 pages; 24 cm edition, Upper Saddle River, NJ, 2010.

[58] R.C. Martin, Clean Coder Blog, 2014. Accessed Jan. 26, 2022, https://blog.cleancoder.com/unclebob/2014/05/08/SingleReponsibilityPrinciple.html.

[59] G. Kim, J. Humble, P. Debois, J. Willis, N. Forsgren, The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations, IT Revolution, 2021.

[60] R. Tighilt, M. Abdellatif, N. Moha, H. Mili, G.E. Boussaidi, J. Privat, Y.-G. Guéhéneuc, On the study of microservices antipatterns: a catalog proposal, in: Proceedings of the European Conference on Pattern Languages of Programs 2020, 2020, pp. 1–13.

[61] S. Newman, Building Microservices, O'Reilly Media, Incorporated, Sebastopol, United States, 2021. http://ebookcentral.proquest.com/lib/etsmtl-ebooks/detail.action?docID=6683096.

[62] J. Paul, Top 10 Microservices Design Patterns and Principles - Examples, n.d. Available online - 6 october 2023, https://javarevisited.blogspot.com/2021/09/microservices-design-patterns-principles.html.

[63] J.A. Scott, A practical guide to microservices and containers, MapR Data Technol. (2018).

[64] M. Hofmann, E. Schnabel, K. Stanley, Microservices best practices for java, IBM Corporation, 2016.

[65] L. community, How do you optimize microservice performance?, 2023, (Community blog). Available online, https://linkedin.com.

[66] M. Bolanowski, Ż. Kamil, A. Paszkiewicz, M. Ganzha, M. Paprzycki, P. Sowiński, I. Lacalle, C.E. Palau, Efficiency of REST and gRPC in realizing communication tasks in microservice-based ecosystems, arXiv:2208.00682 (2022).

[67] J. Mueller, Performance issue considerations for Microservices APIs, 2015, (Blog post). Available online, https://smartbear.com.

[68] R. Dhall, Performance patterns in microservices-based integrations, 2016, (Blog post). Available online, https://dzone.com.

[69] M. Schaefer, 10 Best Practices for Microservices Deployment and Management, 2023, (Blog post). Available online, https://xalt.com.

[70] L. Kumar, Best Practices for Microservices: Building Scalable and Efficient., 2023, (Community blog). Available online, https://dzone.com.

[71] M. Devs, Microservices best practices, 2024, (Whitepaper). Available online, https://mulesoft.com.

[72] S. Gupta, Developing High-Performance Applications Microservices, 2022, (Community blog). Available online, https://medium.com.

[73] P. Raj, S. Vanga, A. Chaudhary, Design, Development, and Deployment of Event-Driven Microservices Practically, 2023, pp. 129–142. https://doi.org/10.1002/9781119814795.ch7

[74] C. Pahl, P. Jamshidi, O. Zimmermann, Architectural principles for cloud software, ACM Transact. Internet Technol. (TOIT) 18 (2) (2018) 1–23.

[75] D. Liu, H. Zhu, C. Xu, I. Bayley, D. Lightfoot, M. Green, P. Marshall, Cide: an integrated development environment for microservices, in: 2016 IEEE International Conference on Services Computing (SCC), IEEE, 2016, pp. 808–812.

[76] A. Parker, D. Spoonhower, J. Mace, B. Sigelman, R. Isaacs, Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices, "O'Reilly Media, Inc.", 2020.

[77] C. Berger, B. Nguyen, O. Benderius, Containerized development and microservices for self-driving vehicles: experiences 'i&' best practices, in: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), 2017, pp. 7–12. https://doi.org/10.1109/ICSAW.2017.56

[78] J. Turnbull, The Docker Book: Containerization is the New Virtualization, James Turnbull, 2014.

[79] G. Sayfan, Hands-on Microservices with Kubernetes: Build, Deploy, and Manage Scalable Microservices on Kubernetes, 2019.

[80] N. Railić, M. Savić, Architecting continuous integration and continuous deployment for microservice architecture, in: 2021 20th International Symposium INFOTEH-JAHORINA (INFOTEH), 2021, pp. 1–5. https://doi.org/10.1109/INFOTEH51037.2021.9400696
[81] M. Hofmann, E. Schnabel, K. Stanley, I. B. M. C. I. T. S. Organization, Microservices Best Practices for Java, IBM Redbooks, 2016. Book.
[82] R. Ranju, How to Improve Performance of Microservices: Best Practices, 2023, (Blog post). Available online, https://sayonetech.com.
[83] S. Bonagiri, 14 Best Practices for Microservices: Boosting Efficiency with DevOps, 2023, (Blog post). Available online, https://passionateprogrammers.com.
[84] M. Waseem, P. Liang, M. Shahin, A. Di Salle, G. Márquez, Design, monitoring, and testing of microservices systems: the practitioners' perspective, J. Syst. Softw. 182 (2021) 111061.
[85] M. Schulz, Bundling and minification: an introduction, 2015.
[86] R. Anderson, Bundle and Minify Static Assets in ASP.NET Core, 2022. Available online - 5 october 2023, https://learn.microsoft.com/en-us/aspnet/core/client-side/bundling-and-minification?view=aspnetcore-7.0.
[87] C. Foundry, Using Blue-Green Deployment to Reduce Downtime and Risk, Year not provided. https://docs.cloudfoundry.org/devguide/deploy-apps/Blue/Green.html#map-green.
[88] V.M. Niño-Martínez, J.O. Ocharán-Hernández, X. Limón, J.C. Pérez-Arriaga, A microservice deployment guide, Program. Comput. Softw. 48 (8) (2022) 632–645. https://doi.org/10.1134/S0361768822080151. https://doi.org/10.1134/S0361768822080151
[89] N.T. Blog, How Netflix scales its API with GraphQL Federation (part 1), Netflix TechBlog (2020). https://netflixtechblog.com/how-netflix-scales-its-api-with-graphql-federation-part-1-ae3557c187e2.
[90] N. Railić, M. Savić, Architecting Continuous Integration and Continuous Deployment for Microservice Architecture, INFOTEH Symposium, 2021. Book.
[91] S. Sengupta, 15 Best Practices for Building a Microservices Architecture - BMC. (Blog post), 2023. Available online, https://bmc.com.
[92] R. Gancarz, LinkedIn adopts protocol buffers for microservices integration and reduces latency by up to 60%, InfoQ 2023 (2023). https://www.infoq.com/news/2023/07/linkedin-protocol-buffers-restli/.
[93] C. Richardson, Microservices Patterns: With Examples in java, Simon and Schuster 2019 .
[94] D.C. Montgomery, Design and Analysis of Experiments, Wiley, Hoboken, N.J., 7th edition, 2009.
[95] S. Newman, Designing Microservices, O'Reilly Media, 2015. Book, http://shop.oreilly.com/product/0636920033158.do.
[96] T. Dyba, T. Dingsoyr, G.K. Hanssen, Applying systematic reviews to diverse study types: an experience report, in: First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), IEEE, 2007, pp. 225–234.
[97] V. Garousi, M. Felderer, M.V. Mäntylä, Guidelines for including grey literature and conducting multivocal literature reviews in software engineering, Inf. Softw. Technol. 106 (2019) 101–121. https://www.sciencedirect.com/science/article/pii/S0950584918301939. https://doi.org/https://doi.org/10.1016/j.infsof.2018.09.006
[98] J. Soldani, D.A. Tamburri, W.-J. Van Den Heuvel, The pains and gains of microservices: a systematic grey literature review, J. Syst. Softw. 146 (2018) 215–232. https://www.sciencedirect.com/science/article/pii/S0164121218302139. https://doi.org/https://doi.org/10.1016/j.jss.2018.09.082