Guide in Designing an Asynchronous Performance-Centric Framework for Heterogeneous Microservices in Time-Critical Cybersecurity Applications. The BIECO Use Case

Rudolf Erdei[1], Emil Marian Pasca[1], Daniela Delinschi[1], Iulia Bărăian[2], and Oliviu Matei[1]

[1]Universitatea Tehnica din Cluj-Napoca - Centrul Universitar Nord din Baia-Mare
[2]Universitatea Tehnica din Cluj-Napoca

March 14, 2023

Abstract

This article presents the architecture, design and validation of a microservice orchestration approach, that improves the flexibility of heterogeneous microservice-based platforms. Improving user experience and interaction, for time-critical applications are aspects that were primary objectives for the design of the architecture. Each microservice can provide its own embedded user interface component, also decentralizing it and, in consequence, improving the loosely coupled approach to the architecture. Obtained results are promising, with high throughput and low response times. Also, a key finding was the introduction of benchmarking as a new step in the development lifecycle of performance-critical software components, with an example of how it can be applied within an Agile methodology. Further research is proposed to improve the results and raise the final technology readiness level of the system. Obtained results already make the approach a candidate and viable alternative to classical service composers.

Received 03 March, 2023; | Revised <day> <Month>, <year>; | Accepted <day> <Month>, <year>

DOI: xxx/xxxx

RESEARCH PAPER

Guide in Designing an Asynchronous Performance-Centric Framework for Heterogeneous Microservices in Time-Critical Cybersecurity Applications. The BIECO Use Case

Rudolf Erdei *[1] | Daniela Delinschi[1] | Emil Marian Pașca[1] | Iulia Bărăian[2] | Oliviu Matei[1]

[1]Technical University of Cluj Napoca, North University Centre of Baia Mare, Romania
[2]Technical University of Cluj Napoca, Romania

Correspondence
*Rudolf Erdei, Email: rudolf.erdei@campus.utcluj.ro

Funding Information
This research was supported by the project BIECO (www.bieco.org) that received funding from the European Union's Horizon 2020 Research and Innovation programme under grant agreement No. 952702.

Abstract

This article presents the architecture, design and validation of a microservice orchestration approach, that improves the flexibility of heterogeneous microservice-based platforms. Improving user experience and interaction, for time-critical applications are aspects that were primary objectives for the design of the architecture. Each microservice can provide its own embedded user interface component, also decentralizing it and, in consequence, improving the loosely coupled approach to the architecture. Obtained results are promising, with high throughput and low response times. Also, a key finding was the introduction of benchmarking as a new step in the development lifecycle of performance-critical software components, with an example of how it can be applied within an Agile methodology. Further research is proposed to improve the results and raise the final technology readiness level of the system. Obtained results already make the approach a candidate and viable alternative to classical service composers.

KEYWORDS:
Service Orchestration, Message-Based Communication, Cybersecurity, Loose Coupling, Edge Computing, Asynchronous Communication, Heterogeneous Communication, Heterogeneous Services, Distributed Systems.

1 | INTRODUCTION

The field of Computer Science and Engineering has undergone significant evolution over the past few decades. The pace of technological advancement is rapid, with innovations and revolutionary systems being launched regularly, as competing creatives vie to be the first to release a specific technology. However, this competitive landscape can sometimes lead teams to cut corners on best practices, especially in terms of *cybersecurity*. As a result, *cybersecurity testing and validation* have become critical issues as the number of vulnerabilities has increased.

Moreover, the introduction of Microservices has further opened up the way for application components to function independently, leading to an increase in vulnerabilities[1]. Microservices are an approach to software development that involves breaking large monolithic applications into smaller modules that can operate independently and be integrated into multiple platforms. While Microservices offer many benefits, such as modularity and scalability, they also present new challenges in terms of security, as these services need to communicate and share data with each other while maintaining the confidentiality and integrity of the data[2]. Thus, ensuring the security and reliability of Microservices is becoming increasingly important in modern software engineering.

In the past couple of decades, microservices have also emerged as a popular alternative to large monolithic systems or inflexible platforms, that also require Rapid Development[3,4,5,1]. Microservices are an improved version of *Service Oriented Architectures* (SOA) and offer several advantages over monolithic applications, such as optimized resource allocation, flexibility, and the ability to integrate *Serverless Components*[6,7,1] and various communication approaches. However, effective and robust communication among microservices requires standardized definitions of several aspects. Typically, service composers or consumers are used to integrating these services, which can result in large codebases and UI components.

The use of Microservices in software development necessitates careful consideration of *Service Composition*[8], *Service Discovery*[9], and *Service Interoperability*[10], all of which require the use of robust framework components. However, *Service Composers* (SC) can quickly become unwieldy, as they must integrate multiple low-level functionalities and also accommodate intuitive user interface and interaction. *System State*[11] is also crucial, as SC's must understand *workflows* and their current state, as well as support simple functionalities like *logging*, *exception handling* and *email sending*. Ultimately, these platform components may themselves become large, monolithic components, introducing limitations into the system, and becoming difficult to modify, extend, or update. As a result, further research is needed to develop more efficient and scalable approaches to service composition in microservice architectures, with the potential use of high-performance message-based communication.

The integration of message-based communication in an SC has the potential to address both the issue of platform security as well as platform efficiency. To address this aspect, a proposed standardization framework for *asynchronous messaging*[12] has been discussed and implemented[13,14,15]. This platform has been implemented within the *Horizon 2020 BIECO Project*[1], a cutting-edge initiative aimed at addressing the challenges of *almost real-time communication* and *high throughput*. Unlike traditional approaches that rely on monolithic components, the proposed framework leverages *orchestration* and *light UI* components to achieve efficient communication in a *hybrid distributed system* (both *local* and *remote*). This approach helps to reduce the complexity associated with traditional SC's, which can be cumbersome to maintain and scale. By utilizing a *modular architecture* that is centred on asynchronous messaging, the proposed framework offers a scalable and robust solution that can adapt to changing user requirements and evolving security threats. Overall, this article provides a compelling argument for adopting a standardized approach to asynchronous messaging in security assessment platforms and demonstrates the potential of the proposed framework to drive innovation and progress in this field.

The remainder of the article is structured as follows: Section 2 structures some of the questions that the work is trying to answer to. Section 3 presents some of the related work that the current platform is based on. Section 4 presents an overview of principles that provide base for our work. Section 5 presents some of the most important details of implementation Section 6 presents some relevant results. Section 7 shows parts of the key findings relevant to the research community. Section 8 presents conclusions related to the current article.

2 | RESEARCH QUESTIONS

During our initial investigation into potential solutions to meet the requirements of the entire BIECO system, we determined that a microservice-based approach would be the most suitable. Due to the inherent complexity and diversity of the individual microservices, certain issues arose, prompting us to formulate a series of research questions:

- **RQ1**: What are the current issues in the usage of heterogeneous microservices, in the context of Loose-Coupling Architectures?
- **RQ2**: How can different communication modes like REST API, JMS, and MQTT be used to improve the integration of heterogeneous services and enable more complex interactions between them?
- **RQ3**: What are the challenges and limitations in designing a decentralized user interface component, and how can they be overcome to create a seamless user experience?
- **RQ4**: How can asynchronous stateful microservices be integrated and how can they be leveraged to enable more complex interactions between services and the platform?

[1]https://www.bieco.org

- **RQ5**: How can System Consistency be assured across all services?
- **RQ6**: What are the best practices for designing and implementing a platform that supports improved Loose-Coupling, heterogeneous types of services, and complex user interface components?
- **RQ7**: How can the platform be designed to support scalability and ensure that it can handle increasing demands from users and services?

3 | RELATED WORK

Software Service Orchestrators are an important part of distributed system design. Centralized orchestration, while having some disadvantages, does introduce a level of abstraction, possible automatization and security that the Internet itself lacks. Zaalouk et al.[16] present an orchestrator architecture for Software Defined Networking (SDN) that can respond to software attacks, raising the security of the resulting network. Another similar approach is presented by Jaeger[17] in a configurable Security Orchestrator approach that can be dynamically configured, thus paving the way to other approaches, like the one presented in this paper.

Distributed Microservices Communication can also be addressed in the context of **Fog Computing**[18]. Orchestrators also play a key role in this kind of system, enabling **Service Composition**, that provide a holistic approach to delivering functionality to the end-users. Brito et al.[19], Davoli et al.[20] and Borsatti et al.[21] have proposed architectures that address some key aspects like awareness, security, compositing, either defining a service component that implements all functionalities, or defining a service layer with different components for each functionality.

Multimodal microservices The architecture of *multimodal microservices* offers a solution to integrate various services developed in different programming languages or technologies. However, this approach comes with its own set of challenges and limitations. The primary concern is inter-service communication, which is usually achieved through RESTful HTTP calls or Advanced Message Queuing Protocol (AMQP). While these solutions can be effective, they can also introduce communication latency, testing challenges, and issues with exception handling. Additionally, IoT or Edge devices may use non-web-friendly protocols such as Bluetooth, ZigBee, or LoRa, making communication more difficult.

In a survey conducted by Gurudat et al.[22], RabbitMQ was found to be more stable and suitable for a large number of users involved in network communication, making it a preferred option over RESTful HTTP calls or AMQP clients. In embedded systems and IoT or Edge devices, communication between these devices often requires overlay networks, tunneling, or dynamic networking architectures, as mentioned by G. Bartolomeo[23]. Lightweight service composition that uses a service proxy can also enable interaction between a web service and an embedded service, as proposed by Wang et al.[24].

Energy-consumption When it comes to scaling various services or allocating (or even relocating the use of) resources, we can definitely also talk about optimized energy consumption, optimal allocation of CPU, bandwidth, or other relevant system resources. In an IoT / Edge Computing environment, all these aspects are relevant and demand utmost consideration both from a research as well as design and implementation points of view. We've seen energy-aware self-adaptation in service-oriented architectures proposition in the work of Frederico et al. [25] and other performance-aware energy-efficient cloud orchestration techniques in Rossi et al. [26] where we see how scale-out applications and setting corresponding sleep state to hosts can minimize the energy consumption.

Transactions Although we take advantage of the decoupling features of microservices, there are also times, when if one or more services fail to accomplish a given task, we must revert to previous actions on the upstream services, as such, as the need for handling with transactions may be required in cases like this, and we saw how it can be accomplished through a **saga pattern** which integrates a quote cache system, this enables various transactions to rollback the changes if they are failed on the cache level, and only the successful transaction will be committed to the database, this technique has been an experiment by Daraghmi et al. [27] similar work related to saga pattern has been also used in the work of Martin et al. [28] for a reactive microservices environment.

Edge-centric IoT architecture poses several challenges due to limited resources and processing power compared to the potentially unlimited capabilities of the cloud for processing Big Data from various IoT devices. Additionally, physical devices requiring real-time responses and handling sensitive data can further complicate the situation. According to a survey by Kewei et al.[29], only a few solutions have been proposed to address these challenges. The survey suggests that a **lightweight secure**

communication solutions with **flexible protocol** usage could be researched further to cope with the edge-centric IoT architecture. In other words, the study recommends exploring a **Distributed Multimodal Microservices** framework to meet this need.

Privacy In an analysis of privacy-preserving constraints in microservices architecture, Vistbakka et al. [30] proposes a lightweight formalization of privacy constraint so that microservices share data on an organization level, and it could also define a subset of data, that could be accessed to another organization through a defined policy. Another popular mechanism for privacy assurance is the well-known Role-Based Access Control (RBAC).

Authorisation and Authentication Depending on the device/protocol used, it may be capable of multi-factor authentication, biometrics, or using a simple way as using a username/password method, despite the method used, end-to-end security must be achieved, and ideas on how this could be achieved in the past were by verification object in Pang et al. [31] or by certificates, as mentioned by Dai et al. [32], and blockchain solutions proposed by Zhu et al. in [33,34]. As far as BIECO, the platform itself should impose the way microservices authenticate within the platform, so as not to introduce unnecessary complexity levels, that would in the end reduce performance.

Orchestrating IoT devices and services is a challenge on its own, as the IoT ecosystem has several hard limitations that cannot be pushed. Wen et al.[35] discuss these limitations and issues, proposing a Fog Orchestrator that aims to address them, while also proposing future research trends for this specific area.

Scalability, **Interoperability**, and **Reliability** are terms frequently discussed when it comes to orchestrating microservices. The need for an efficient way to handle these aspects, especially in the context of IoT and edge computing applications is discussed by Shi et al.[36] in their paper on the challenges of Edge Computing. To address these problems Jamborsalamati et al.[37] proposed an Autonomous Resource Allocation system with a hierarchical architecture, and also local and global communication layers based on MQTT and HTTP TCP/IP protocols for cloud interactions. Another Publish-Subscribe architecture is also proposed by Saif et al.[38] who proposed an HTTP/3 (H3) solution that exploits the wide-ranging improvements made over H2 and takes better advantage of **QUIC** transport than an **MQTT** mapping would.

To reduce development costs and also provide a centralized solution for security systems based on Service Oriented Architecture (SOA), Goutam et al.[39] proposed an architectural framework called **Orchestrator Model for System Security** (OSS) which is based on three components: Services, Security Services and an Orchestrator.

4 | ARCHITECTURAL CHOICES AND COMMUNICATION PRINCIPLES IN HYBRID MICROSERVICES-BASED ARCHITECTURE OF BIECO

The BIECO architecture (see Figure 1) includes two main connected service types: *local* and *remote*. The integration of various types of Services within one system is an essential aspect that enables flexibility in the platform's functionality. By integrating multiple types of Services, the system is not limited to available Services for one specific type of connection. Additionally, several types of connection protocols are available (e.g.: REST, Java JMS, MQTT), making it crucial to design the system to allow management across many protocols. This type of design further enhances the system's flexibility and opens up new possibilities for the platform's future development.

Moreover, the integration of local and remote Services is critical in enabling the platform to operate in diverse environments and accommodate future integration of Services. For instance, when dealing with remote Services, the system must address several challenges like *network latency*, *low bandwidth*, and other *connectivity issues*[40]. In contrast, local Services offer higher speed and reliability but may lack in keeping information, models and versions up to date (it becomes the responsibility of the user to keep the software components up to date). The platform's ability to integrate both types of Services allows the system to leverage the advantages of each type and compensate for the other's shortcomings. This flexibility also allows for future expansions and updates to the platform's functionality, enabling it to adapt to changing requirements.

In addition to its location-based design, the BIECO platform's ability to manage multiple connection protocols is a critical aspect that increases its adaptability. By integrating several types of protocols, the platform can provide reliable connectivity and communication between the different Services within the system, even when faced with varying network conditions. This type of adaptability is essential, especially in scenarios where the platform needs to operate in diverse environments, with varying degrees of connectivity and network speeds.

When developing the BIECO Service Communication System, multiple considerations were taken into account[41] including the *types of Services* and *communication modes*. The system accommodates both *synchronous* and *asynchronous* Services,

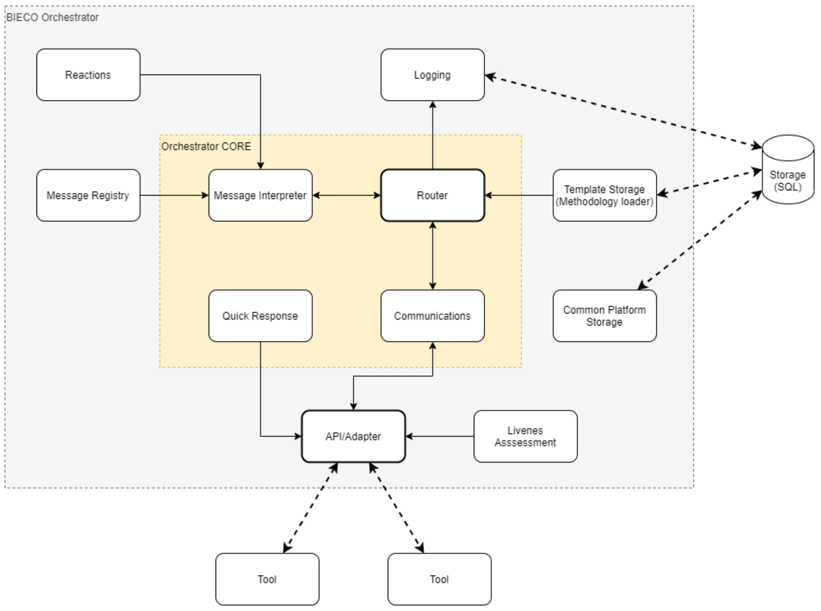

FIGURE 1 High Level BIECO Orchestrator Architecture[14]

also supporting *stateful* and *stateless* ones. Additionally, *Events* and *Data* types that flow through the system were abstracted into standardized *Messages*[13,42] to provide maximum flexibility for various requirements. The design also focused on *platform performance*, *scalability*, *flexibility*, and *extensibility*, resulting in different abstractions, formalizations, and standardizations.

Regarding the medium of choice and communication standards, *REST APIs*, *ActiveMQ*, and *Mosquitto* (MQTT) were considered. To mitigate component failures, a *Heartbeat* mechanism is used to poll the system state, and a *Quick Response* mechanism is designed to halt the system in case of component failure. For critical components, this fail-safe mechanism can be used to protect real hardware or systems. Furthermore, *data integrity*, *safety*, and *security* were also considered. To ensure *message validity*, a complex message validation pipeline was designed and implemented as discussed in detail in[15].

The integration of local and remote Services within the BIECO platform, along with the management of multiple connection protocols, is an example of how flexibility and adaptability can be achieved in a system. This design philosophy allows the platform to operate seamlessly in a variety of environments, thereby catering to the diverse requirements of end-users. With the rapidly evolving landscape of modern technologies and services, the ability to adapt to new functionality and capabilities is paramount. The BIECO platform's flexible architecture can enable it to easily integrate new services seamlessly, reducing the development time and complexity involved in upgrading the platform. Moreover, the platform can be easily scaled up or down, depending on the specific use-case and requirements, making it an ideal choice for both small-scale and large-scale applications.

The flexible architecture of the BIECO platform provides a level of future-proofing, ensuring that the platform remains relevant and functional in the long run. By design, the platform is capable of accommodating new services and technologies, thereby future-proofing it against potential obsolescence. Furthermore, the platform's adaptability and flexibility also translate into cost savings and faster turnaround times. Since the platform can be quickly adapted to new services and technologies, the need for additional resources or time-consuming redesigns will be significantly reduced. As a result, the platform can be easily maintained and upgraded, resulting in a cost-effective and efficient solution for a variety of applications.

5 | PLATFORM IMPLEMENTATION DETAILS

The architecture for the BIECO Orchestrator has been presented in depth in[14]. This platform is designed to execute asynchronously in order to achieve near-realtime responsiveness and be non-blocking for the microservices. This is especially important given one of the *Execution Methodologies* (EM) utilized for the *System Under Test* (SUT), as described by Calabro et al. and Daoudagh et al.[43,44]. This EM requires real-time live data, which must be analyzed in the context of a Predictive Simulation implemented through a Digital Twin, as also mentioned by Cioroaica et al.[45].

An asynchronous platform is a system that allows for the execution of tasks without the need for continuous interaction (with the user or with the services) or instant responses (like in the case of classical REST APIs). Instead, the system can process requests in the background while users continue with other activities. Asynchronous platforms typically rely on messaging protocols to communicate between components and can be designed to handle high-throughput scenarios where processing large amounts of data is critical. One key advantage of asynchronous platforms is their ability to handle unpredictability and variability in the workload, as they can queue up tasks and execute them as resources become available. This makes them ideal for distributed systems that require robustness, fault tolerance, and scalability, like the case of the BIECO platform.

The Services that are executed within the BIECO platform use a custom **lifecycle** that was thoroughly described in [14]. Each Service is considered a *state machine* that can and should manage its own status. Polling the Service's status is done via a *Heartbeat* mechanism, managed by the Orchestrator. There are two intervals of polling, one for the case when the system is idle, with an interval of 30s, and another one when the system is in execution, with an interval of 10s. The Heartbeat mechanism's role is to assess the state of each executing Service, in order to assess if a Project can be executed or to be able to deploy safety and security mechanisms if needed.

Within their execution lifecycle, BIECO Services can request interaction with the user, so the messaging system provides a mechanism for displaying a custom UI to the user, which will require some input. The platform itself will wait for the user to finish the interaction and continue with normal execution. This kind of interaction provides a higher level of flexibility, greatly improving the interaction possibilities between the user and the platform components.

In order to meet platform requirements, the entire design, implementation, and software stack of the BIECO Orchestrator were carefully chosen and optimized to minimize overhead and maximize efficiency. Various acceleration techniques, such as *In-Memory Cache Management* and *ontological modelling of platform behaviour*, were used in conjunction with low-level code optimizations like *benchmarking data-structure choices* and *optimizing algorithms* used throughout the platform. These techniques combined greatly enhance the resulting codebase, making it as efficient and responsive as possible.

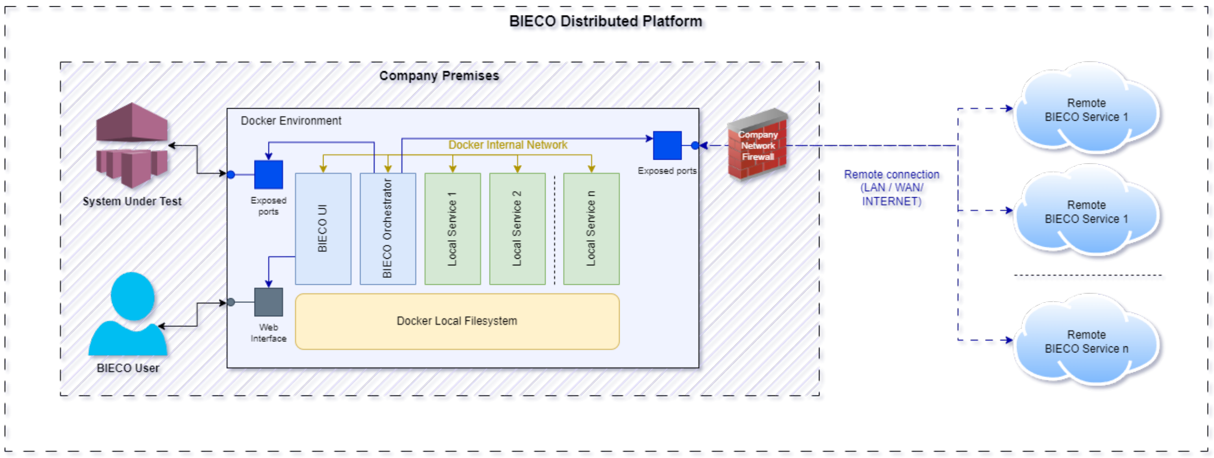

FIGURE 2 Deployment Architecture for the BIECO Distributed Platform

Another architectural choice was the deployment of software components as close as possible to the SUT (see Figure 2). In this regard, *Docker*[2] has become a popular technology in software engineering and is widely used in the deployment and delivery of microservices. It allows for the creation and management of containers that are isolated from each other and from the host system, making it easier to package and distribute applications. In the context of the BIECO project, Docker is being used as a containerization platform to package and deploy the various microservices and other components of the system. This includes the BIECO Orchestrator, BIECO UI, containerized storage and database access and file transfer between local Services./ This approach allows for greater flexibility and agility in managing the deployment and configuration of the platform, as each component can be packaged, configured and deployed independently, and can be easily scaled up or down as needed. Docker also provides a consistent and reproducible environment for running the system, ensuring that all components run in the

[2]https://www.docker.com/

same way across different machines and environments. Docker also allows for easier maintenance and updates of the system, as each container can be updated independently without affecting the rest of the system.

Furthermore, Docker is also used in the development process of the BIECO platform, as it enables developers to create a consistent environment across different machines and operating systems. This allows for more efficient testing and deployment of the platform, ultimately leading to a more stable and reliable system for users, that can use the platform in its entirety, or only use the components they require, without the need to install all the available Services. Accessing the BIECO platform is done via the BIECO UI, implemented as a web service so as to not limit installation of BIECO only to GUI-enabled operating systems. Internal web servers are used to expose UI components to the user via browsers installed on any machine within the Local Area Network (LAN). This ensures low latency, as internet connectivity will usually slow down the response times of any Service.

The use of Docker also helps to improve the security and reliability of the platform. By isolating each component within its own container, the risk of contamination or interference between different components is minimized. In addition, the use of container images ensures that each component is deployed in a consistent and repeatable way, reducing the risk of configuration errors or other issues that could impact the functioning of the platform. Docker also provides built-in security features, such as isolation between containers and secure communication between components, which can help to mitigate the risk of security breaches and data leaks. Overall, the use of Docker in the BIECO project helps to provide a more flexible, reliable, and secure platform for conducting security assessments.

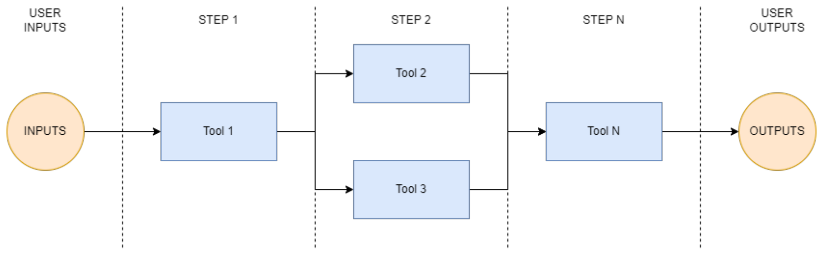

FIGURE 3 Toolchain of BIECO Services (example of Methodology)[14,13]

In summary, the BIECO Orchestrator has been designed with a focus on *high-performance* and *near-realtime response*, in order to support the Execution Methodologies (see Figure 3) utilized for the System Under Test. With careful consideration of the software stack and optimization techniques used, the resulting codebase is able to provide the required *flexibility*, *efficiency* and *responsiveness* for this demanding application.

5.1 | Testing the platform from a Cybersecurity perspective

Although the advantages of microservices are well-known, there are also complex challenges that arise from microservices orchestration and asynchronous communication. These challenges include potential human errors, security concerns at every layer of the application, and the need for specific security procedures for each component. For example, developers must consider security measures for the application layer, such as authentication and authorization, as well as for the communication layer, such as message encryption and network security protocols. Failure to address these challenges could lead to vulnerabilities and pose serious risks to overall system security.

In the article by Mateus-Coelho et al.[40], the topic of Microservice Security is discussed from different points of view, all of them regarding the system as a logical upscale of a monolith. Communication in this case is done directly from one microservice to another, lacking the capabilities that a flexible Loose-Coupling would introduce. The article describes security threats and measures that could be taken into consideration, all with the aim of building a defense system around microservices themselves, as well as the system as a whole. Although not all were applicable in our case, we have introduced some of the measures suggested, and also many others that fit the case of BIECO better.

The cybersecurity design and execution methodology employed for the BIECO platform's primary components involve the following actions from our side:

- Despite the fact that the platform will be shielded by the network firewall, it is not dependent on it and treats the internal LAN as an open Internet environment. This approach is also consistent with the fact that certain ports can potentially be opened to allow external microservices to access the BIECO platform;
- We made sure that the software libraries offer support and mitigation for the most common software vulnerabilities (like XSS, SQL injection, and others);
- We designed an atypical lightweight but secure *Authentication* system, that enables fast Service authentication, but is based on behaviour, so that the security of the platform itself will not be compromised if an outside attacker would somehow gain access to the authentication tokens;
- We used both internal and external tools to analyze the source-code and library versions for known and potential vulnerabilities (e.g.: by using public CVE/CVSS databases, but also tools like SonarQube, Snyk);
- We used established and maintained libraries for common operations like authentication, JSON encoding and others;
- We used containerization strategies to separate the components of the BIECO platform from the rest of the local operating system, in order to abide by internal company rules and protect the local environment from potential security flaws within the BIECO platform;
- In order to protect against *Denial-of-Service* (DoS) attacks, we have developed a system that only accepts requests related to the currently executing project. By enforcing compliance with platform requirements during service testing, any request that falls outside the project context is flagged as potentially malicious and can be blocked without impacting system performance. In order to protect against *Denial-of-Service* (DoS) attacks, we have implemented the system only to accept requests related to the currently executing project. By enforcing compliance with platform requirements during Service testing, any request that falls outside the project context is flagged as potentially malicious and can be blocked without impacting system performance.
- In order to safeguard the system from potential data leaks and ensure *Data Security*, encryption was employed. Network traffic was captured using tools such as Wireshark or Nmap, and the data was analyzed to ensure that no sensitive information could be read.
- We incorporated *Secure Development*[46,47,48] practices, such as secure coding, code reviews, and regular security testing, to ensure the practice of secure development.

The BIECO platform, as a whole, is designed to work within the end user's premises. This implies a strict security policy that companies usually set up for themselves. BIECO abides by these rules, making sure not to open security problems in the end user's intranet.

6 | RESULTS

The BIECO platform in its entirety has a resulting TRL of 6 and has been used to demonstrate three diverse use-cases. All use-cases share a distributed nature, one being in the automotive industry (secure remote software updates), one in the energy industry (reading heterogeneous energy sensing devices) and one in fin-tech (a machine learning platform for transactions).

Every use-case had the following limitations:

- *Source code* is regarded as secret and cannot leave the premises of the company. No person outside the company can view this source-code;
- Interaction with the SUT should be as simple as possible to implement, also generated Events and Data should be agnostic to receivers;
- Using the BIECO Platform should be agnostic to the implementation details of the use-case.

These limitations were addressed on two levels:

1. At the Platform level: deployment within the Docker environment resulted in installing the platform at the user's premises, so no source-code has to leave the company. This also resulted in greatly improving the platform's response time;

2. At the Service level: Services define a broker-based abstract messaging system, fully integrated with the BIECO Orchestrator, that involved a very small code that needed to be integrated inside the system (called a `Probe`[44]).

By these two simple choices, the BIECO system is able to be fully agnostic to any implementation details (like the used programming language, software stack, architectural design, and other aspects). Execution within the testing environment has yielded the following results:

TABLE 1 Execution statistics for BIECO Platform components

BIECO Component (Service name)	Average Response Time (ms)	Messages Throughput (msgs/s)	Mean Time To Discovery (s)	Exception Fail-safe Triggering (ms)
BIECO Orchestrator	11.3	1323	30 / 10	10
BIECO UI	121.6	532	N/A	70

An important metric regarding the use of Services is *Mean Time to Discovery* (MTTD), defined as the average time it takes to discover a problem in the microservice. This metric, within BIECO, defines how fast the Orchestrator discovers changes in Service states or potential problems. In the case of BIECO, this metric is actually hard-coded, and implemented within the Heartbeat mechanism. As previously stated (see Section 5), there are two situations, when the platform is idle (MTTD of 30s) and when the platform is executing a Project (MTTD of 10s). The UI will only assess the connection to the Orchestrator at 30s intervals. Also, a `WebSocket` connection is open whenever a Project is opened, which provides continuous assessment for the state of the Orchestrator.

The experimental setup used to obtain the results involved a powerful *Ryzen 5 4600H* processor with a generous amount of 64 GB of *RAM* and a *Windows 10* operating system. The *Docker* environment was installed with *WSL2*, and to optimize its performance it had access to 4 out of 6 internal cores of the physical processor. The system's *RAM* usage was carefully monitored during the experiments and was found to be around 8GB when the system was idle, while in full load, it reached an average of 20GB. The platform services (the UI and Orchestrator) were using an average of 100-400 MB, depending on the throughput of Messages. The processor usage was also monitored and found to be under 1% when idle (despite the constant *Heartbeat* assessment), and between 25% to 80% when the platform was under full load. This variation depended on the complexity of the Services that were being executed, especially those that included ML engines with large models, requiring more processing power. Overall, this experimental setup provided a solid foundation for obtaining the results presented in this study, showcasing the platform's performance under realistic conditions.

The results are in line with the central idea of BIECO and all its requirements. While still needing deeper analysis and further testing, they indicate a successful implementation of the BIECO Service Communication System and its associated Services, the current TRL level brings it close to being a viable solution for real-world scenarios. This provides a solid foundation for the continued development of the platform and its integration with external systems, leading to increased efficiency, adaptability, and scalability. As the platform evolves and additional features are added, ongoing testing and refinement will be necessary to ensure its continued success and reliability, with the aim of raising the TRL to a required 9, in order to be market-ready.

7 | KEY FINDINGS

Through intense work, benchmarking and testing various components and code snippets, BIECO has undergone a strict spiral of implementation, rather novel and adapted to this kind of project (focused on cybersecurity and performance). It has become a flexible and adaptable platform that enables the integration of local and remote Services, allowing it to operate in diverse environments and adapt to changing requirements, due to its intrinsic flexibility. The platform's design takes into consideration different types of services and communication modes, as well as events and data types that will flow through the system, and uses different abstractions, formalizations, and standardizations to maintain performance, scalability, flexibility, and extensibility. BIECO's asynchronous architecture is designed to provide near-real-time responses, which is important for runtime assessment of the System Under Test and for the analysis of real-time live data.

Docker is used as a containerization technology to manage different dependencies and ensure that the platform runs consistently across different environments. BIECO leverages Docker's features, like `Docker Volumes`, `Docker Networks`, and `Docker Applications`, to provide a smooth and transparent experience to the user. Docker enables BIECO to be a portable high-performance and highly-flexible solution that is hardware and operating system agnostic.

The results of performance testing show that the platform can handle multiple services running simultaneously, with low latency and high throughput, while keeping resource consumption at reasonable levels, by today's hardware standards. The BIECO platform has been developed as part of the BIECO research project, in the fields of cybersecurity, cyber-physical systems, Industry 4.0. BIECO has been tested in various and diverse use cases, including automotive, machine learning cloud applications and data-collection/data-fusion. The platform is still evolving, with ongoing research and development focused on improving its capabilities, adding new functionalities, and enhancing its usability and user experience.

7.1 | Benchmarking as an Important Step in Software Development Cycle

While working on the BIECO platform and researching ways of optimizing the platform performance, we found out that *Benchmarking* was a very important step in the implementation cycle of any software component that is focused on performance and security. The role of benchmarking is two-fold:

1. Understanding and optimizing critical software components and pieces of code (functions, libraries), so that the component will use the least resources while offering the maximum throughput of information;
2. Testing the component from a security perspective, to understand the impact of various validation patterns and methods over performance.

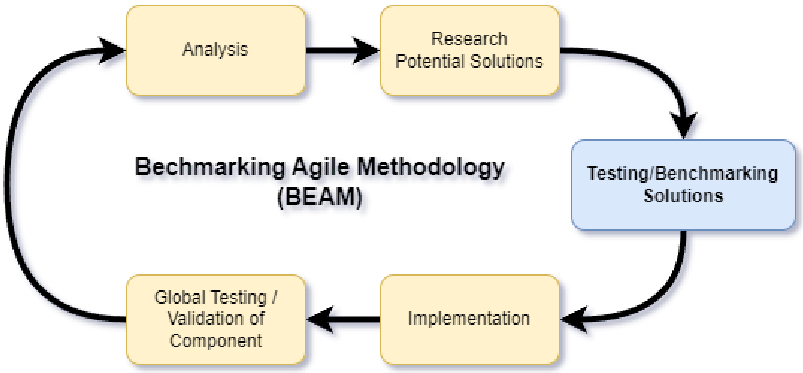
Analysis
Research Potential Solutions
Bechmarking Agile Methodology (BEAM)
Testing/Benchmarking Solutions
Global Testing / Validation of Component
Implementation

FIGURE 4 Depiction of steps and placement for the Benchmarking step within the BEAM Methodology

The results that we obtained by formalizing `benchmarking` as a step in the development lifecycle, were at least interesting. Contrary to some "Best Practices" and "Code Writing Patterns/Guidelines", we had to merge some functionalities that traditionally were performed by different classes, into a single class (e.g.: Validating entities and validating inputs). This required a new level of creativity in order to preserve code maintainability and also a low `Technical Debt`, without compromising on performance. This was an iterative process, as the requirements for the platform itself were sometimes changing. An Agile methodology was used as a base, with an iteration of 4 weeks, and had the following steps (see also Figure 4):

- Analysis of requirements within the already implemented codebase;
- Researching possible solutions that would address these requirements (at least three of them);

- Testing (benchmarking) the possible solutions and observing various parameters, like: memory usage, computational effort, possible vulnerabilities introduced, possibility to abstract and extend the functionalities, and possibility to merge with existing classes;
- Implementation of the chosen solution;
- Testing and small-scale validation of the resulting codebase.

Rather than being a Greedy approach, where the chosen solution was never revisited (if the requirements do not change), we treated everything within a backtracking approach: at each iteration, the modified or affected modules were benchmarked again, so as to ensure the optimal performance of the platform. We have formalized this development lifecycle methodology within the team with the proposed name of *Benchmarking Agile Methodology* (BEAM).

7.2 | Lessons Learned

The advantages and disadvantages of this approach were presented in[14]. However, there are some lessons (more abstract, not-so-technical ideas) that we learned from this project that are important enough to be mentioned:

- Services are encouraged to develop their own interaction UI's. A decentralized UI simplified development for both the UI Integrator, as well as the Service development. Simple mechanisms for facilitating interaction can be enough to generate a complex, intuitive UI with a good User Experience. This reduced the development time for the platform with at least 30%, by not needing to define and implement special situations and validations;
- A flexible starting architecture, that solves most of the requirements from the initial steps of the lifecycle, is of utmost importance. This will offer the most flexibility while also assuring that `Technical Debt` is kept as low as possible. This offered the maximum amount of flexibility later on when requirements changed drastically;
- Innovation must be done while also leveraging current advancements. Testing multiple solutions is very important, while also being open to newer, *not yet established* solutions. The easiest-to-implement solution might not be the best. By using Docker virtual environment as opposed to a real environment (or a centralized one), we were able to solve multiple problems at once, like network delays, the availability of source code to internal services, parallelisation of Services (to be able to run more Projects in parallel).
- Innovation should not be only in the technical areas. Small changes within the work style can have huge benefits. By integrating the benchmarking step within the lifecycle, we were able to maximize both security as well as performance in only one step.

Using the REST HTTP standard for requests has the benefit of being easily adaptable and is also readily available in all major programming languages. Standard errors, caching, proxies and payloads can have formats such as JSON, XML or other. HTTP also supports *Asynchronous Communication*, an important characteristic that is needed by many of the Services used in the BIECO Project.

8 | CONCLUSIONS

We have divided this section into two parts, one for adressing the RQ's that were presented in Section 2, and one for the final conclusions regarding the presented work

8.1 | Addressing the Research Questions

8.1.1 | RQ1. Current Issues:

As presented in Sections 1 and 3, the issues include communication heterogeneity and microservice heterogeneity (local/remote, stateful/stateless), which may lead to complex execution login on the orchestration part and also lack of flexibility for the platform.

8.1.2 | RQ2. Communication Modes Integration:

As presented in Section 5, *abstraction* is the key for integrating virtually any communication mode. Asynchronous communication relies on atomic units of data, abstracted as Messages and implemented in a unified way in all communication modes. With small adaptations, this abstraction is applied to all microservice and communication types.

8.1.3 | RQ3. Decentralized User-Interface:

A poorly designed UI may not attract users, while an overly complex one may require extensive development time and also a significant amount of runtime resources. As stated in Section 5, we implemented a flexible UI, with two modes of integration for each Service: either through required inputs that get rendered within the UI, but also by providing a mechanism that will display the Service standalone UI, at demand. This allowed for a much simpler implementation with much more flexibility.

8.1.4 | RQ4. Integrating Asynchronous Stateful Microservices:

Stateful microservices have the benefit of not requiring a live connection to wait for Service finalization. This introduces a level of complexity in the execution logic but optimizes the required connections, as well as the resources consumed by the orchestration service. The design of a high-throughput and secure system requires the implementation of certain abstractions and conventions, as discussed in Sections 4 and 5, which are dependent on the types (and potential types) of communication modes used.

8.1.5 | RQ5. System Consistency:

Researching related work, but also from our own work within BIECO, *System Consistency* is achieved when the orchestration service has a correct and updated overview of all the current states of microservices. This is achieved, in part, as presented in Section 5, with the `Heartbeat` mechanism, and with well-established microservice states.

8.1.6 | RQ6. Best Practices:

These best practices include *benchmarking* (Section 7.1) for assuring system performance, *abstractions* (Section 4) for assuring consistency and scalability, *conventions* (Section 5) for assuring a solid and valid base for the platform to be built upon.

8.1.7 | RQ7. Scalability:

We found that optimizations and benchmarking are very important aspects of development from the scalability aspect. This will result in a well-performing platform even on low-end hardware. Dynamic scalability is obtained by using containerization approaches, as presented in Section 7.1.

8.2 | Final thoughts

The article discusses a novel approach to heterogeneous microservices orchestration. The resulting system provides support for local and remote services, REST API, JAVA JMS (through ActiveMQ) and MQTT (through Mosquitto) data connections. The heterogeneity is designed into the platform itself by abstracting the communication and data streams with the `Message` entity. By using the Docker containerization solution, the system is fully scalable. Also, by dockerizing each component separately, the system can be distributed to several interconnected machines, with the only limitation being the availability of a TCP connection.

When comparing our platform to other approaches such as Zaalouk et al.[16], Jaeger[17], and others (as discussed in Section 3), significant differences can be observed. These differences include:

- Strong focus on service heterogeneity and acceptance of more generic connection types (like REST in an asynchronous context);
- Abstracting communication across connection types, via the *Message* entity, that can be safely implemented in any communication mode;
- Improved Loose-Coupling, allowing services to function both independently or as part of the platform;
- Heterogeneous types of services (local and remote) and communication modes (REST API, JMS, MQTT);
- Ability to design complex non-standardised user interface components for all services, which can be effortlessly integrated;

- Integration of service states, enabling more complex interactions between services themselves and with the platform.

The presented BIECO platform underwent testing both in a controlled laboratory environment consisting of 10 microservices, as well as validation with three use-cases (automotive, energy, and fin-tech, as presented in Section 6). During the testing phase, the interaction comprised a 5-step methodology, with 1 to 5 services operating in parallel at each step. The system functioned correctly during testing, making it a strong candidate for further validation and demonstration in more challenging environments that are closer to the final deployment environment. Validating with the use-cases involved one or two methodologies per use-case, with at least 5 Services per methodology. More complex methodologies involved the use of 10 Services, both local and remote. Services were executed both in parallel, with the correct exchange of Data and Events, as well as part of a sequential toolchain. This validation step provided a more accurate assessment of the platform's overall performance, reliability, and suitability for deployment in real-world scenarios.

During the testing of the Methodology Driven Orchestration Service, some limitations were identified. One issue was observed when Services emitted Events at inappropriate times, leading to confusion for the Orchestration Service. Additionally, the system encountered difficulties when managing a sudden surge of data, such as when multiple Services emitted a large number of messages with large payloads simultaneously. This resulted in network overload and memory constraints on the virtual machine. Both these issues were mitigated in validation by improving the Methodology Execution Engine, the Message Invalidation Engine, and by using more processors for the Orchestrator virtual machine. Also, a transaction-based Methodology Execution Engine will be researched, for solving potential temporary issues with communication. This would further raise the TRL of the resulting platform, improving the scaling capabilities.

Current as well as future research and development efforts are focusing on optimizing system responses and improving behavioural modifications in the case of invalid messages from Services, to increase system resilience and expand the range of problems it can address. Other aspects of improvement that will be addressed include the detection of malicious access attempts using machine-learning approaches, designing a more complex module for non-standard APIs (like public APIs), and integrating Distributed Learning Services (based on Federated Learning).

Conflict of interest

The authors declare no potential conflict of interest.

References

1. Matei O, Skrzypek P, Heb R, Moga A. Transition from Serverfull to Serverless Architecture in Cloud-Based Software Applications. In: Springer. ; 2020: 304–314.

2. Kritikos K, Skrzypek P, Moga A, Matei O. Towards the modelling of hybrid cloud applications. In: IEEE. ; 2019: 291–295.

3. Agarwal R, Prasad J, Tanniru M, Lynch J. Risks of rapid application development. *Communications of the ACM* 2000; 43(11es): 1–es.

4. Beynon-Davies P, Carne C, Mackay H, Tudhope D. Rapid application development (RAD): an empirical review. *European Journal of Information Systems* 1999; 8(3): 211–223.

5. Martin J. *Rapid application development*. Macmillan Publishing Co., Inc. . 1991.

6. Matei O, Erdei R, Moga A, Heb R. A Serverless Architecture for a Wearable Face Recognition Application. In: Springer. ; 2021: 642–655.

7. Matei O, Materka K, Skyscraper P, Erdei R. Functionizer-A Cloud Agnostic Platform for Serverless Computing. In: Springer. ; 2021: 541–550.

8. Jula A, Sundararajan E, Othman Z. Cloud computing service composition: A systematic literature review. *Expert systems with applications* 2014; 41(8): 3809–3824.

9. Mukhopadhyay D, Chougule A. A survey on web service discovery approaches. In: Springer. ; 2012: 1001–1012.

10. Bouzerzour NEH, Ghazouani S, Slimani Y. A survey on the service interoperability in cloud computing: Client-centric and provider-centric perspectives. *Software: Practice and Experience* 2020; 50(7): 1025–1060.

11. Primadianto A, Lu CN. A review on distribution system state estimation. *IEEE Transactions on Power Systems* 2016; 32(5): 3875–3883.

12. Matei O, Erdei R, Delinschi D, Andreica L. Data Based Message Validation as a Security Cornerstone in Loose Coupling Software Architecture. In: Springer. ; 2021: 214–223.

13. Delinschi D, Erdei R, Matei O. Ontology Driven High Performance Messaging System for Distributed Software Platforms. In: IEEE. ; 2022: 1–6.

14. Erdei R, Delinschi D, Pașca E, Matei O. Orchestrator architecture and communication methodology for flexible event driven message based communication. In: Springer. ; 2022: 127–137.

15. Matei O, Erdei R, Delinschi D, Andreica L. Data based message validation as a security cornerstone in loose coupling software architecture. In: Springer. ; 2022: 214–223.

16. Zaalouk A, Khondoker R, Marx R, Bayarou K. OrchSec: An orchestrator-based architecture for enhancing network-security using network monitoring and SDN control functions. In: IEEE. ; 2014: 1–9.

17. Jaeger B. Security orchestrator: Introducing a security orchestrator in the context of the etsi nfv reference architecture. In: . 1. IEEE. ; 2015: 1255–1260.

18. Yi S, Hao Z, Qin Z, Li Q. Fog computing: Platform and applications. In: IEEE. ; 2015: 73–78.

19. De Brito MS, Hoque S, Magedanz T, et al. A service orchestration architecture for fog-enabled infrastructures. In: IEEE. ; 2017: 127–132.

20. Davoli G, Cerroni W, Borsatti D, Valieri M, Tarchi D, Raffaelli C. A Fog Computing Orchestrator Architecture with Service Model Awareness. *IEEE Transactions on Network and Service Management* 2021.

21. Borsatti D, Valieri M, Tarchi D, Raffaelli C. A Fog Computing Orchestrator Architecture with Service Model Awareness. 2021.

22. S GK, T P. A Better Solution Towards Microservices Communication In Web Application: A Survey. *International Journal of Innovative Research in Computer Science & Technology* 2019.

23. Bartolomeo GD. Enabling Microservice Interactions within Heterogeneous Edge Infrastructures. In: ; 2021.

24. Wang S, Du C, Chen J, Zhang Y, Yang M. Microservice Architecture for Embedded Systems. In: . 5. ; 2021: 544-549

25. Oliveira A. dFG, Ledoux T. Self-Optimisation of the Energy Footprint in Service-Oriented Architectures. In: GCM '10. Association for Computing Machinery; 2010; New York, NY, USA: 4–9

26. Rossi FD, Xavier MG, De Rose CA, Calheiros RN, Buyya R. E-eco: Performance-aware energy-efficient cloud data center orchestration. *Journal of Network and Computer Applications* 2017; 78: 83-96. doi: https://doi.org/10.1016/j.jnca.2016.10.024

27. Daraghmi E, Zhang CP, Yuan SM. Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture. *Applied Sciences* 2022; 12(12). doi: 10.3390/app12126242

28. Štefanko. M, Chaloupka. O, Rossi. B. The Saga Pattern in a Reactive Microservices Environment. In: INSTICC. SciTePress; 2019: 483-490

29. Sha K, Yang TA, Wei W, Davari S. A survey of edge computing-based designs for IoT security. *Digital Communications and Networks* 2020; 6(2): 195-202. doi: https://doi.org/10.1016/j.dcan.2019.08.006

30. Vistbakka I, Troubitsyna E. Analysing Privacy-Preserving Constraints in Microservices Architecture. In: ; 2020: 1089-1090

31. Pang H, Tan KL. Authenticating query results in edge computing. In: ; 2004: 560-571

32. Dai J, Alves-Foss J. Certificate based authorization simulation system. In: ; 2001: 190-195

33. Zhu Y, Wu X, Hu Z. Fine Grained Access Control Based on Smart Contract for Edge Computing. In: ; 2022.

34. Zhu Y, Huang C, Hu Z, Al-Dhelaan A, Al-Dhelaan M. Blockchain-Enabled Access Management System for Edge Computing. *Electronics* 2021; 10(9). doi: 10.3390/electronics10091000

35. Wen Z, Yang R, Garraghan P, et al. Fog Orchestration for Internet of Things Services. *IEEE Internet Computing* 2017. doi: 10.1109/mic.2017.36

36. Shi W, Cao J, Zhang Q, Li Y, Xu L. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 2016; 3(5): 637–646. doi: 10.1109/JIOT.2016.2579198

37. Jamborsalamati P, Fernandez E, Moghimi M, Hossain MJ, Heidari A, Lu J. MQTT-Based Resource Allocation of Smart Buildings for Grid Demand Reduction Considering Unreliable Communication Links. *IEEE Systems Journal* 2019; 13(3): 3304–3315. doi: 10.1109/JSYST.2018.2875537

38. Saif D, Matrawy A. APure HTTP/3 Alternative to MQTT-over-QUIC in Resource-Constrained IoT. In: ; 2021: 36–39

39. Goutam A, Rajkamal , Ingle M. Orchestrator Model for System Security. In: Unnikrishnan S, Surve S, Bhoir D. , eds. *Advances in Computing, Communication and Control*Springer Berlin Heidelberg; 2011; Berlin, Heidelberg: 195–199.

40. Mateus-Coelho N, Cruz-Cunha M, Ferreira LG. Security in Microservices Architectures. *Procedia Computer Science* 2021; 181: 1225-1236. CENTERIS 2020 - International Conference on ENTERprise Information Systems / ProjMAN 2020 - International Conference on Project MANagement / HCist 2020 - International Conference on Health and Social Care Information Systems and Technologies 2020, CENTERIS/ProjMAN/HCist 2020doi: https://doi.org/10.1016/j.procs.2021.01.320

41. Jamshidi P, Pahl C, Mendonça NC, Lewis J, Tilkov S. Microservices: The journey so far and challenges ahead. *IEEE Software* 2018; 35(3): 24–35.

42. Elmhadhbi L, Karray MH, Archimède B, Otte JN, Smith B. PROMES: An ontology-based messaging service for semantically interoperable information exchange during disaster response. *Journal of Contingencies and Crisis Management* 2020; 28(3): 324–338.

43. Daoudagh S, Marchetti E, Calabrò A, et al. An Ontology-Based Solution for Monitoring IoT Cybersecurity. In: Springer. ; 2022: 158–176.

44. Calabrò A, Cioroaica E, Daoudagh S, Marchetti E. BIECO Runtime Auditing Framework. In: Springer. ; 2022: 181–191.

45. Cioroaica E, Daoudagh S, Marchetti E. Predictive Simulation for Building Trust Within Service-Based Ecosystems. In: IEEE. ; 2022: 34–37.

46. Howard M, Lipner S. *The security development lifecycle*. 8. Microsoft Press Redmond . 2006.

47. Lipner S. The trustworthy computing security development lifecycle. In: IEEE. ; 2004: 2–13.

48. Futcher L, Solms vR. Guidelines for Secure Software Development. In: SAICSIT '08. Association for Computing Machinery; 2008; New York, NY, USA: 56–65