IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 10 August 2025, accepted 25 August 2025, date of publication 2 September 2025, date of current version 10 September 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3604775

RESEARCH ARTICLE

Performance Evaluation of Monolithic and Microservice Architectures for Natural Language Processing in Command and Control Applications

FLAVIO FERREIRA DA SILVA MOSAFI[1], MATEUS SZE COSENZA[1], LÉO VICTOR CRUZ VASCONCELOS[1], LUÍS FERREIRA PIRES[2], (Member, IEEE), JULIO CESAR DUARTE[1], AND MARIA CLAUDIA REIS CAVALCANTI[1]

[1]Military Institute of Engineering, Rio de Janeiro 22290-270, Brazil
[2]University of Twente, 7522 NB Enschede, The Netherlands

Corresponding author: Flavio Ferreira Da Silva Mosafi (flavio.ferreira@ime.eb.br)

This work was supported in part by the National Funds through FINEP—Financiadora de Estudos e Projetos and FAPEB—Fundação de Apoio à Pesquisa, Desenvolvimento e Inovação do Exército Brasileiro, through the Project "Sistema de Sistemas de Comando e Controle" (Reference no: 2904/20), under Contract 01.20.0272.00; and in part by the Air Force Office of Scientific Research under Award FA9550-22-1-0475.

ABSTRACT The success of Command and Control (C2) operations relies heavily on clear communication among participants, often under stressful and time-sensitive conditions that demand accurate decision-making. Examples of such operations include military engagements, reconnaissance missions, search and rescue efforts, and peacekeeping activities. The design of systems to support C2 is therefore challenging, as they must support structured communication and interaction means among participants. To address this, we developed a method called Method to Support Semantic Interoperability in Command and Control (MAISC^2) to support C2, in which we have applied Natural Language Processing (NLP) and semantic techniques to pinpoint specific elements in sentences, thereby enhancing the participants' understanding of the C2 communications. Microservices Architecture (MSA) are known to offer potential benefits over MA, particularly in terms of scalability, independence, and maintenance. Furthermore, some NLP applications have already been developed using MSA, confirming these benefits. This paper presents a version of our system developed using MSA and compares it to its Monolithic Architecture (MA) counterpart. Our objective is to evaluate whether the benefits of adopting MSA, observed in other domains, also apply to C2 support systems based on NLP and semantic techniques, as exemplified by the MAISC^2 method. Our results show that MA outperforms MSA in many scenarios; however, as the application load increases, MSA shows increasingly better performance. In particular, although MA had shorter processing and delivery times under normal conditions, MSA delivered better processing performance when handling significantly higher levels of concurrent activity.

INDEX TERMS Command and control, microservice architecture, military communication, named entity recognition, natural language processing, semantic interoperability.

I. INTRODUCTION

The success of Command and Control (C2) operations depends on the clear understanding among the participants, often in stressful situations, which should allow them to make proper decisions [1]. Examples of C2 operations include military, reconnaissance, search and rescue, and peacekeeping, among others [2]. The design of systems to support C2 is then challenging, since they have to support these participants by offering structured communication and interaction means. We have developed a method called MAISC^2 [3] to support C2, in which we have applied Natural

The associate editor coordinating the review of this manuscript and approving it for publication was Engang Tian.

Language Processing (NLP) and semantic techniques to pinpoint specific elements in sentences, aiming at enhancing the understanding of the C2 operation participants.

Machine Learning (ML), which is a branch of Artificial Intelligence (AI), employs diverse learning strategies to enable systems to learn and perform tasks based on data. Therefore, developing efficient meta-heuristic approaches to enhance the quality of input data for training ML models can significantly improve learning outcomes and model performance [4]. NLP is a task that can employ ML to assist agents in better understanding messages by applying specialized techniques for handling textual data, such as Syntactic Analysis Parsing, Part-Of-Speech tagging (POS), and Named Entity Recognition (NER) [5]. The rapid growth in the use of NLP across all businesses enabled the analysis of large volumes of text data more swiftly and assertively to identify hidden knowledge and behavior patterns. NER plays an important role here because it aims to identify entities with specific meanings in the corpus, such as name, place, organization, etc [6]. It is extensively utilized across a wide range of industries, enhancing the efficiency and accuracy of information processing by automatically extracting essential data from unstructured text. For instance, in healthcare, Named Entity Recognition (NER) supports physicians in swiftly accessing patient histories and relevant medical literature. In finance, it facilitates the analysis of financial news and strengthens anti-money laundering efforts [7]. It is the fundamental technique for downstream applications, including information extraction, knowledge graph construction, machine translation, and question-answering systems [8]. In addition to using NLP, the software architecture can have an impact on the performance, scalability, and maintenance of the C2 systems, making them more suitable to support C2 operations.

In general, applications designed to support C2 operations must be prepared to deal with adverse environments, such as in the case of a military operation. In this sense, a physical or cyber attack can compromise functionality and jeopardize the operation. Rapid scalability also poses a significant challenge, particularly for MA. In military contexts, operational dynamics such as the integration of additional military or civilian units, the need to incorporate new functionalities, or the urgency of maintenance driven by evolving mission requirements demand flexible and adaptable system architectures. In Mosafi et al. [3], we presented a software system based on NLP developed to support the MAISC^2 method, originally designed and implemented using a Monolithic Architecture (MA). However, MSA are known to bring potential benefits over MA, especially concerning scalability, independence, and maintenance [9]. Furthermore, some NLP applications have already been developed using MSA, confirming these benefits [10]. Therefore, this work presents a version of our system implemented using MSA and compares it to its MA counterpart. The main objective is to evaluate whether the benefits of adopting MSA, observed in other domains, also apply to C2 support systems based on NLP and semantic techniques, as exemplified by the MAISC^2 method. While previous works have explored the benefits of MSA in various domains, including e-commerce, healthcare, and IoT, we did not find prior studies that specifically combine these aspects in C2 applications.

Unlike prior studies that focus on general performance metrics, architecture migration, or microservice identification in domains such as e-commerce or IoT, our contributions are threefold:

1) **NLP-Focused Pipeline Evaluation**: evaluating the performance of microservices versus monolithic implementations in the context of a real NLP-based application, with analysis of each pipeline stage (e.g., entity recognition, message delivery);
2) **Statistical Microservice**: introducing a dedicated statistics microservice that collects operational metrics without interfering with the processing pipeline; and
3) **C2-Specific Use Case**: applying these architectural strategies specifically to C2 systems, where message processing under stress and operational clarity are essential, expanding the applicability of MSA research to an underexplored domain.

The paper is further structured as follows. Section II presents the theoretical foundation of this work in terms of the basic concepts. Section III presents the details of our proposed method and the system architecture, explaining our main design choices. Section IV presents our experimental results, where we compare the MA and MSA versions of our system. Section V discusses related work, highlighting previous studies and solutions while identifying gaps that this work aims to fill. Finally, Section VI summarizes our findings and conclusions and provides recommendations for future work.

II. THEORETICAL FOUNDATION

This section presents the fundamental concepts that support this study. It begins by outlining the principles of MSA and MA, followed by a discussion of key concepts related to NER.

A. MICROSERVICES VS MONOLITHIC

Since this paper aims to compare the performance of versions of an application developed with different software architectures (Monolithic versus Microservices), it is essential to begin by conceptually distinguishing these two architectures. A Monolithic Architecture (MA) consists of a software design in which all system components, such as user interface, business logic, and data source, are tightly integrated and deployed as a single unit [11]. This architecture becomes progressively harder to maintain and extend, resulting in scalability and organizational challenges [36]. One of the benefits of this approach is that it is relatively easy to understand and manage, particularly for simple systems and smaller teams or projects, due to its straightforward and

centralized implementation structure. In addition, in the MA code, reusability can be achieved when adopting a modular programming style within the same system [12].

However, challenges can arise when multiple application components are invoked concurrently; for example, scalability may become a limiting factor [13]. The modernization of legacy systems has become a focal point for many organizations. The microservices are emerging as a promising solution to improve them [14]. Modern software applications are expected to handle high levels of concurrency and availability while remaining scalable, well-structured, and minimally interdependent. Along with the increasing technological complexity, the scope of these applications has expanded over time. New features, often outside the original business domain, have been incorporated, contributing to the system's growth and its overall complexity [15].

In contrast to MA, a Microservices Architecture (MSA) is a design approach where an application is built as a collection of relatively small, autonomous services that interact through well-defined Application Programming Interface (API) [16]. Each of these services has a specific purpose with a well-defined role, focusing on relatively small tasks that solve a particular application problem [16]. In essence, an MSA embodies a modular decomposition of the system in which each service is responsible for a specific business capability, is independently deployable, and often operates within its own process. This approach offers several benefits, including improved scalability, technological flexibility, independent deployment, fault isolation, and enhanced maintainability.

Any software can be modularized and designed as an MA or an MSA. However, MSA offers more flexibility, as each module is executed as an independent service and can be deployed on separate hosts, allowing one service to fail or undergo maintenance without impacting others. Another benefit is that each module may require different resources, such as more memory or processing nodes. They may also require different technologies, such as a lightweight Database Management System (DBMS) for efficiency reasons, or a more robust system for handling complex operations.

Moreover, with the increasing popularization of MSA, many projects that initially adopted MA later transitioned to MSA to overcome scalability and maintainability issues. This evolution shows the importance of comparing these two architectures to understand their trade-offs and identify specific scenarios in which each architecture may offer distinct advantages. Table 1 presents a comparative overview of MSA and MA, highlighting their main characteristics.

TABLE 1. Comparative overview of MSA and MA.

Aspect	MSA	MA
Communication	APIs or messaging	Internal function calls
Deployment	Independent services	All components deployed together
Development	Harder for large systems	Easier for small systems
Failure	A service failure usually affects itself	A single failure can cause the system crash
Maintenance	Easier to manage and update services independently	May require the whole system to be stopped
Scalability	Easier to scale individual services	Hard to scale each part separately
Structure	Collection of small, independent services	Single, unified application
Technology	Each service can use different technologies	Often limited to one language or framework

B. NAMED ENTITY RECOGNITION

Textual information can be encountered daily, often carrying deeper meaning beyond its literal form. Depending on the context, the same word can have different meanings. A task widely used to classify the terms of a sentence is Named Entity Recognition (NER), which supports disambiguation. Jayatilleke et al. [5] defines NER as an NLP task designed to extract relevant information from textual documents. The purpose of NER is to identify certain elements in the corpus as a named entity and all its mentions. It can be divided into two sub-tasks, namely identifying labels and boundaries. NER is generally associated with entity categories that carry distinct meanings or strong referential properties within a given corpus. These entities serve as basic elements in heterogeneous text sources, enabling structured information extraction and analysis [7].

Although NER can be used to highlight an important term in the text, it can also be used for other purposes in NLP tasks, like categorizing terms from a given class, such as personal names, place names, and organization names [6]. In addition to the standard entity categories provided by NER applications, other entities can be defined for particular information domains. In such cases, the model should be retrained to accommodate these domain-specific definitions [17]. Furthermore, it is possible to create a dictionary of words that, when combined with other methods, can enhance the performance of the model being trained and generated. Several works have adopted Machine Learning (ML) based approaches to perform this task [18]. NLP relies on annotated text examples to create a training set, which in turn generates a language model capable of recognizing and classifying entity occurrences in new, unseen texts. In C2 operations, NER can be used to highlight key terms that are important for proper decision-making. For example, in situations where an agent has limited experience or is unfamiliar with specific terminology, by highlighting relevant entities, attention can be drawn to essential information, thereby improving situational awareness and supporting more informed decisions. According to Daneshfar et al. [37], it is possible to extract information from unstructured data using approaches such as deep learning and text clustering. This approach can be used in various applications, such as sentiment analysis, recommender systems, and others. Its

accuracy is typically linked to two components: dimensionality reduction and clustering.

III. MAISC2 IMPLEMENTATIONS

This section introduces the MAISC^2 method and its monolithic and microservices-based implementations.

A. THE METHOD

The MAISC^2 method was designed to assist C2 agents in better understanding text messages within a chat room, enhancing their ability to read and interpret information. It comprises two stages: (1) preparation and configuration, culminating in the creation of a trained model called *C2 Knowledge*, and (2) application of the *C2 Knowledge* model, which comprises four tasks (Entity Recognition, Priority Classification, Message Delivery, and Statistics Generation), as illustrated in Figure 1.

The Entity Recognition (ER) task focuses on identifying C2 entities within a message. It utilizes a predefined set of categories (see Table 2), such as Unit (UNT) and Action (ACT), to annotate the message by associating the identified entities with these categories. The Priority Classification (PC) task assesses each message to determine whether it should be prioritized or not. Based on this evaluation, the task annotates the messages accordingly. All these annotations serve as inputs to the Message Delivery (MD) task, which formats the message elements using colors, symbols, and other features to enhance and enrich the presentation of the message. A fragment of the MAISC^2 application interface is shown in Figure 2, depicting examples of messages in a chat room, with annotated categories and prioritized messages colored in red. Finally, the Statistics Generation (SG) task gathers and stores operational data for future use, which can be used to train new versions of the *C2 Knowledge* model, facilitating continuous improvement and adaptation. Currently, the model has been trained on an initial limited dataset, which implies that additional data is required to improve its overall performance.

To provide a clearer understanding of how the *C2 Knowledge* model works, Figure 1 depicts a message exchange between two actors, one acting as an operator and the other as the commander, within a military operation using C2 as a communication approach. In this system, every outgoing message is processed by the C2 Knowledge model, which is integrated into the message exchange platform. In this scenario (Figure 1), the operator sends a message to the commander, reporting the sighting of an enemy platoon and requesting authorization to engage. Upon message transmission, the model initiates the Entity Recognition task, identifying relevant C2-named entities. In the example, in Figure 1, it detects three entities: Enemy and Platoon as Unit [UNT], and North as Direction [DRT], from a predefined list of categories, which is shown in Table 2. Following this, the Priority Classification task determines the urgency of the message by comparing it with previously sent but still unread messages. The message is flagged as a priority due to its nature as a request, although other prioritization rules may apply, such as specific combinations of entity types. The Message Delivery task formats the message to enhance readability and presentation for the recipient, in this case, the commander. Finally, the Statistics Generation task captures and stores metadata about the message in JavaScript Object Notation (JSON) format, facilitating interoperability with other systems and enabling future analysis.

In addition to the categories shown in Table 2, others can be defined. The MAISC^2 Categories serve as the basis for helping agents better understand messages, as they correspond to key terms relevant to C2 operations. One potential issue in exchanging textual C2 messages is the large volume of communication, which, depending on the agent's expertise, can vary in effectiveness regarding situational awareness. Therefore, highlighting the terms from each category in the messages theoretically helps the agent comprehend the content more quickly. In our system, we selected the following categories: Action, Direction, Device, Event, Place, Player, Supplies, Unit, Vehicle, and Weapon. Each of these categories is represented by specific terms that, when identified in a message, are labeled accordingly to enhance the agent's understanding and support informed decision-making. In addition, each category is associated with an acronym that simplifies the annotation process by making it easier to label and identify relevant terms within the message. Examples of terms and their corresponding acronyms are: the Action category, which is denoted by the acronym ACT and includes terms such as *attack*, *retreat*, *rescue*, and *communicate*, and the Player category, represented by the acronym AGT, which includes terms like *policeman*, *doctor*, *sergeant*, and *consultant*. Table 2 gives an overview of the selected categories, along with their acronyms and representative terms.

TABLE 2. C2 entity categories.

Category	Acronym	Example list of possible terms
Action	ACT	attack, retreat, rescue, communicate
Direction	DRT	north, south, east, west
Device	DVC	notebook, walkie-talkie, antenna, flashlight
Event	EVT	fire, storm, mission, rescue
Place	PLC	plain, neighborhood, avenue, region
Player	AGT	policeman, doctor, sergeant, consultant
Supplies	SPL	food, medicine, ammunition, water
Unit	UNT	headquarters, platoon, squadron
Vehicle	VHC	airplane, car, motorcycle, tank
Weapon	WEP	pistol, bayonet, knife, machine gun

As previously mentioned, MSA is an approach for developing a suite of small, independent services that work together as a single application [19] and communicate with each other through API [20]. To ensure a fair comparative analysis, the same MAISC^2 application was implemented in

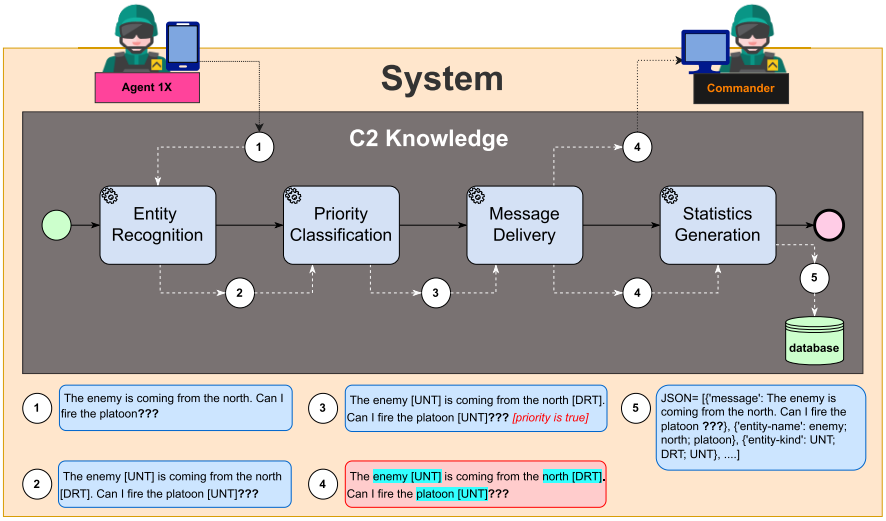


FIGURE 1. MAISC^2 method, using *C2 Knowledge*.

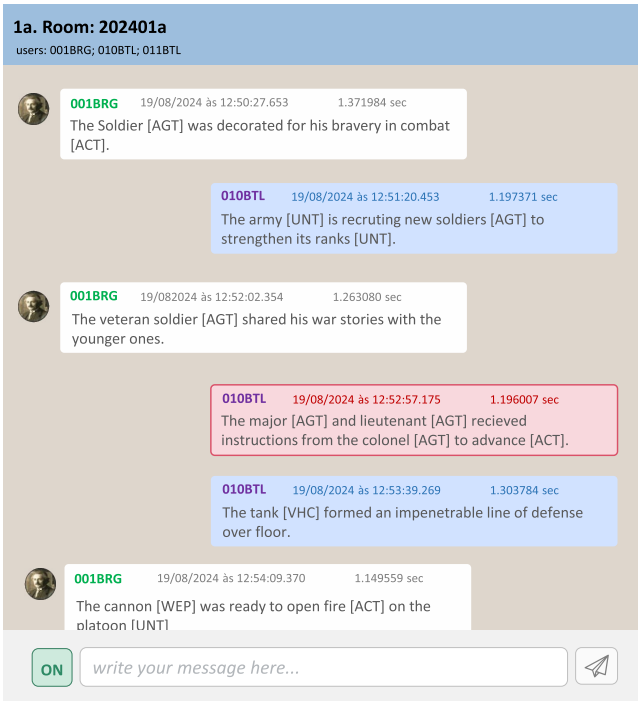


FIGURE 2. MAISC^2 system chat application. Illustration of message exchange in a chat room.

the MA and MSA versions. Accordingly, the MA version was deployed as a single, standalone software, while the MSA version consists of software components that run on distinct hosts.

B. MONOLITHIC ARCHITECTURE

In a Monolithic Architecture (MA), the software is designed so that different components, such as authorization, business logic, and notification modules, are combined into a single, unified program on a single platform [21]. In the MA version, the entire MAISC^2 application runs on a single Virtual Machine (VM) hosted on a physical machine. The architecture comprises a single unit, namely the Application, depicted in Figure 3.

The monolithic version of the application was implemented using Django and PostgreSQL. Django was chosen for its maturity and rapid development features, which are well-suited for prototyping complex web-based systems. PostgreSQL was selected due to its robustness, support for complex queries, and compatibility with Django, making it an appropriate option for managing structured data in a centralized architecture.

The application was divided into two modules, one containing the user interface and the other containing the business logic and data, as shown in Figure 3. The user interface corresponds to the client software through which the users access the application. This module was implemented using Bootstrap, jQuery, and Python. The communication between the user interface module and the business module has been implemented using WebSocket channels, where a WebSocket is a communication protocol commonly employed in instant messaging applications. The business module is responsible for the four MAISC^2 tasks, performing the whole message processing, from the Entity Recognition up to the Statistic Generation task. In this module, the components are hosted in Docker containers. Additionally, the application uses Django to manage data interactions between inputs and the database through business logic, while PostgreSQL is used as the database server.

C. MICROSERVICES ARCHITECTURE

Since we developed the monolithic MAISC^2 in a modular way by employing structured programming, it was possible to identify potential business domain boundaries for microser-

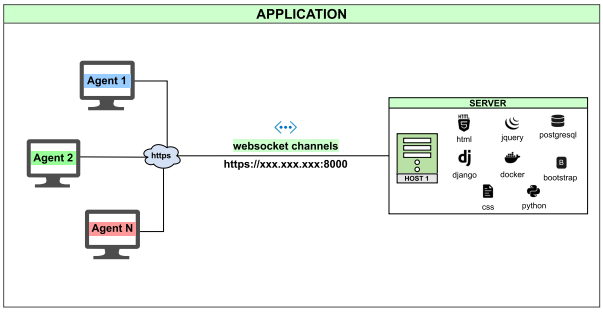


FIGURE 3. MAISC^2 monolithic architecture.

vices. Therefore, the initial step of migrating MA MAISC^2 to MSA was quite straightforward, in which a microservice was defined for each business domain. Due to the pipeline nature of the MAISC^2 method, each task of the method feeds into the next task, so the MSA version followed the pipeline that is shown in Figure 1, and each pipeline task is performed by a microservice, which consists of the application Back-End in Figure 4.

Each microservice was deployed on a server, containerized using Docker, and made accessible through Representational State Transfer (REST) APIs. Besides the microservices that correspond to the method tasks, a central microservice was defined to orchestrate them, which operates within the so-called Central Application (CA). The CA is the connection point with the user interface located on the Front-End. The User Interface represents the clients through which users interact with the business logic.

The assignment of tasks to different microservices not only enabled the use of different database technologies but also addressed the challenge in MA of maintaining and updating shared databases [22]. For instance, the Entity Recognition module uses SQLite as a database software due to its lightweight nature, while the CA relies on PostgreSQL for its robustness. Additionally, the Entity Recognition module handles detailed data related to message annotation, including processing time to identify named entities, the number of tokens, tokens representing entities, types of entities, and their positions within the message, the total number of entities, and the date and time they were processed. In contrast, the CA does not require detailed persistent storage, since it focuses solely on logging the processed messages.

The decision to use SQLite for lightweight services such as NER, and PostgreSQL for more complex modules, like the orchestrator, was driven by performance and resource considerations. SQLite is serverless and therefore has relatively low overhead and fast read/write access, which is beneficial for services requiring quick, localized storage without concurrent access. Conversely, PostgreSQL provides advanced concurrency control and robust transaction handling, making it more suitable for centralized coordination and logging, as required by the orchestrator. This heterogeneity introduced challenges for ensuring consistency across services. To mitigate this, we designed each microservice to manage its own data independently, applying eventual consistency where strict synchronization was not relevant.

IV. EXPERIMENTS AND RESULTS

This section evaluates the performance of the MA and MSA versions of the MAISC^2 application. This experiment follows the same hardware, software, and procedures reported in [3]. Both MAISC^2 versions have been deployed in two different scenarios to draw useful and objective comparative results about their performance. More specifically, these deployments were created using Docker Compose. Each VM is configured with Ubuntu 22.04.3, 64 GB RAM, 120 GB HD, and 16 cores (8 sockets, 2 cores per socket). The physical machines are equipped with an Intel Xeon CPU E7-4870, 2.40 GHz. The experiment consisted of two distinct scenarios:

1) **Scenario 1**: A single VM hosts all services within only a container, representing a monolithic setup.
2) **Scenario 2**: A distributed environment is used, and each microservice runs on an independent VM. These VMs are distributed across two different physical machines.

These scenarios were chosen to simulate the two most common architectural deployments encountered in real-world system design. Scenario 1 mirrors a cost-effective, centralized deployment often used during initial development phases or in resource-constrained environments, while Scenario 2 represents a scalable, production-level setup that takes advantage of distributed computing to enhance fault isolation, load balancing, and horizontal scalability. These scenarios were designed to enable a fair comparison of the MA and MSA implementations of the same application.

For the experiments, a data source with three hundred and sixty messages related to C2 was selected for exchange among three participants. Two neutral people with no contact with the development of the MAISC^2 application were chosen to play the role of agents in an operation, and one person, who participated in the development of the application, was a supervisory agent. Each user played a hierarchical role, where the supervisor assumed the role of commander and the neutral users the role of subordinates. An important detail for conducting the experiments was that the messages that each participant sent to the chat room (Figure 2) depended on the receipt of a message sent by another participant in the same chat room, simulating an actual conversation flow. The experiment lasted approximately 1 hour and 30 minutes, and users accessed the application remotely from their respective workplaces. The metrics chosen for this work over others, such as latency and reliability, were prioritized because they directly impact the dynamics of message exchange in C2. These metrics reflect the speed with which information is processed and the results are made available to users. Latency is inherently captured within processing and delivery times,

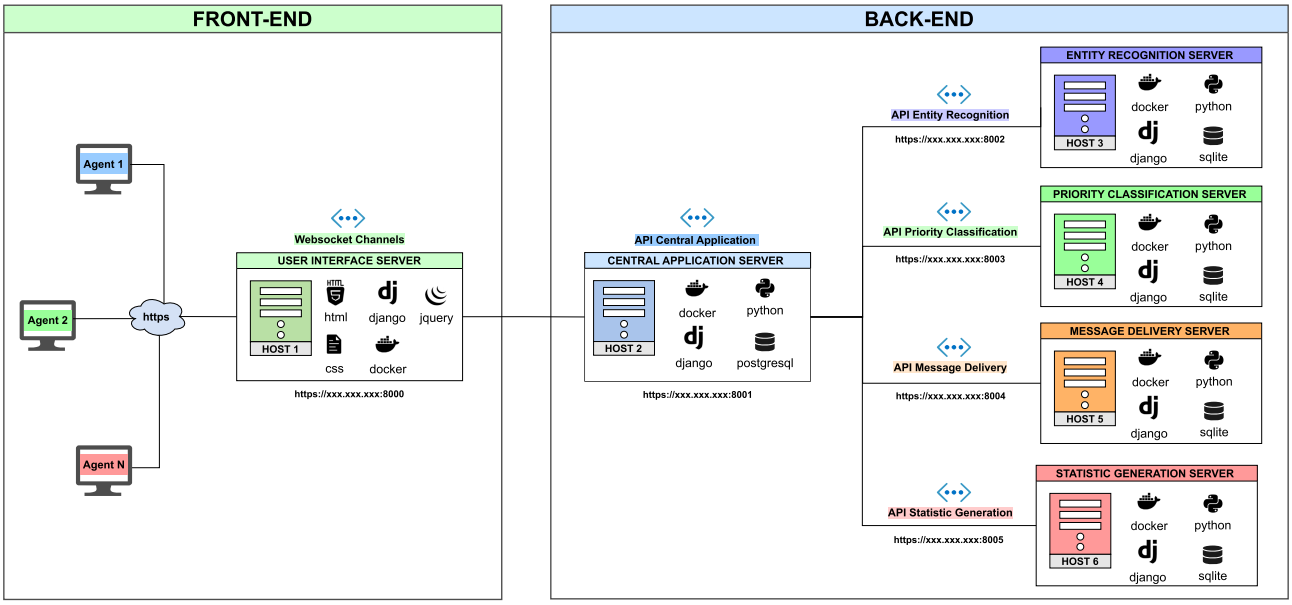


FIGURE 4. MAISC^2 microservices architecture.

as it affects the system's overall responsiveness. Reliability is an essential aspect, but it was outside the scope of the study since the main objective was to compare the performance of the different architectural choices.

The performance of the MA and MSA versions was evaluated based on several key metrics, focusing on processing and delivery times for different tasks. The results for each task are the following:

1) **Processing Time**: This assesses the time required to process messages. The results indicate that the MA approach demonstrated superior performance, with an average processing time approximately 30 milliseconds shorter than that of MSA (Figure 5).
2) **Delivery Time**: This refers to the time taken to deliver messages between agents. On average, MA achieved a delivery time approximately 2 seconds faster than MSA (Figure 6).
3) **Entity Recognition Processing Time**: This evaluates the efficiency of entity recognition in both systems. The MA approach exhibited a lower average processing time compared to MSA, indicating more efficient entity recognition (Figure 7).
4) **Priority Classification Processing Time**: This involves the time required to determine whether a message is classified as a prty. The results show that MA outperformed MSA, achieving faster classification on average (Figure 8).
5) **Message Delivery Processing Time**: This considers the time spent formatting and transmitting messages. The average delivery processing times for MA and MSA were comparable, with no statistically significant difference observed (Figure 9).
6) **Statistic Generation Processing Time**: This refers to the time required to generate statistical outputs. In this aspect, MSA exhibited better performance, surpassing MA in average processing time (Figure 10).

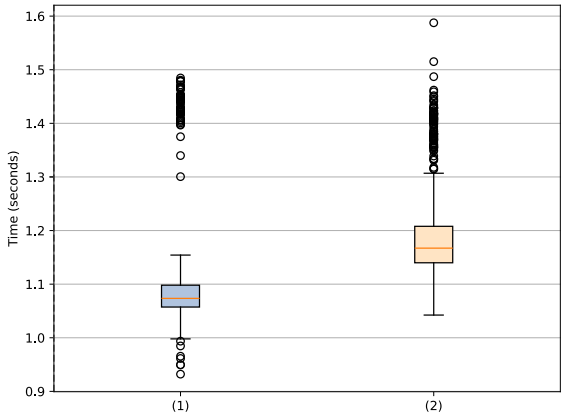


FIGURE 5. Average Processing Time: (1) MA; (2) MSA.

Although MA had shorter processing and delivery times under normal conditions, MSA delivered better processing performance when handling significantly higher levels of concurrent activity. In the MA implementation, all components of the application pipeline operate within a single process on the same machine. As a result, simultaneous processing of multiple requests leads to resource contention, especially for disk and network access, since all stages compete for shared resources. In contrast, the MSA version distributes the application pipeline across independent

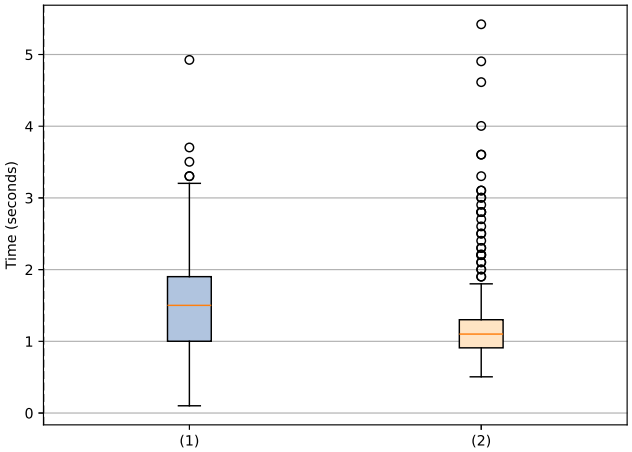


FIGURE 6. Average Delivery Time: (1) MA; (2) MSA.

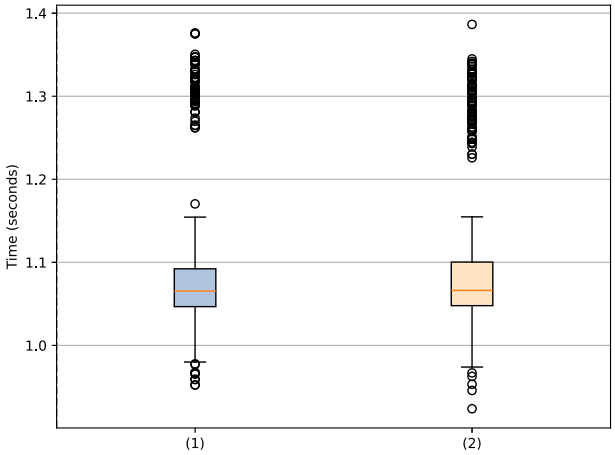


FIGURE 7. Entity Recognition Processing Time: (1) MA; (2) MSA.

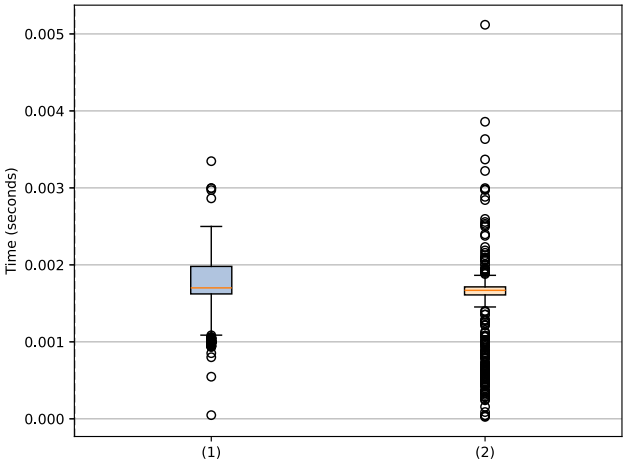


FIGURE 8. Priority Classification Processing Time: (1) MA; (2) MSA.

services, each running on a separate VM. This deployment allows each microservice to handle its own data and commu-

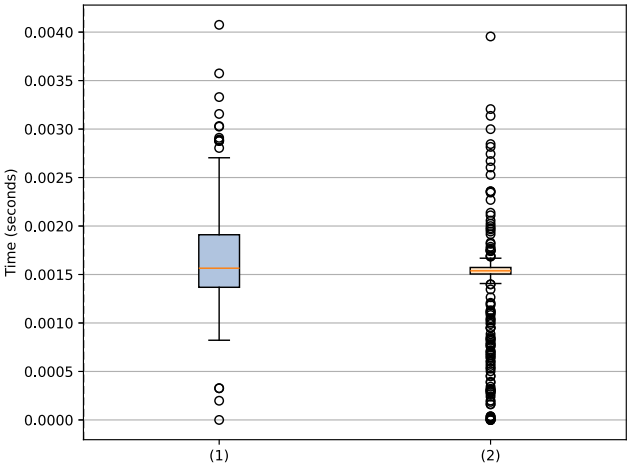


FIGURE 9. Message Delivery Processing Time: (1) MA; (2) MSA.

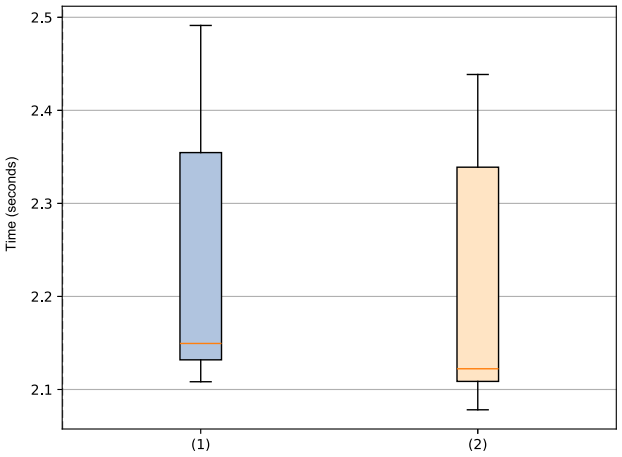


FIGURE 10. Statistics Generation Processing Time: (1) MA; (2) MSA.

nication, reducing contention and distributing the workload across different hardware resources. These differences are reflected in the varying slopes of the performance graphs.

Tests were also conducted on the processing and delivery times of user messages to determine if physical location or other factors might cause variations that could affect the overall averages obtained in the global tests. In terms of processing time, there was a slight variation observed with one of the users (user2) concerning the most recent messages in the MSA tests; however, the average processing time remained consistent at 1.19 seconds per message (Figure 11). A similar pattern was also observed for the message delivery time following processing (Figure 12).

We also compared the processing times for each pipeline task (Entity Recognition, Priority Classification, Message Delivery, and Statistic Generation) in both the MA and MSA versions. As previously discussed, in MA, each step that corresponds to a microservice in MSA is implemented as a function. These functions are invoked by a central controller

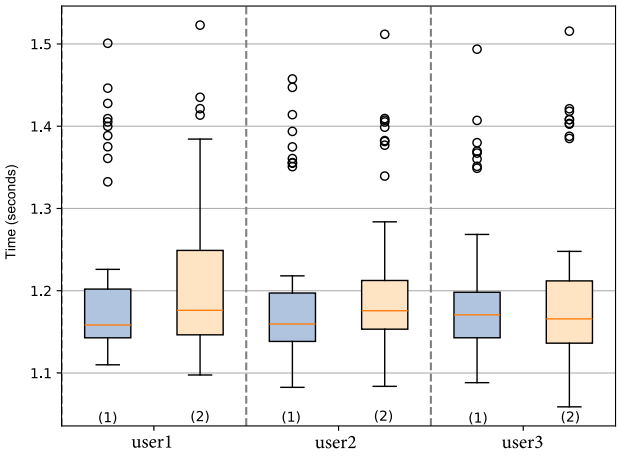


FIGURE 11. Average processing time by user. (1) MA; (2) MSA.

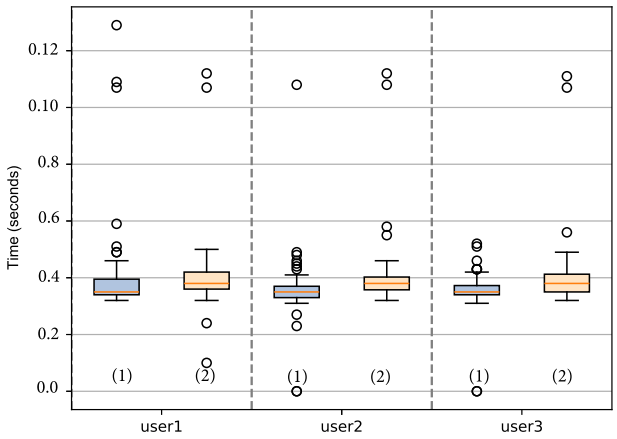


FIGURE 12. Average delivery time by user. (1) MA; (2) MSA.

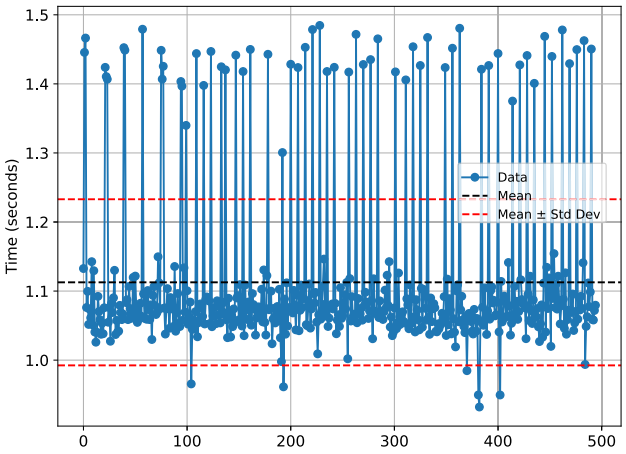


FIGURE 13. MA - Processing time with mean and standard deviation.

whenever execution is required, and they return values based on their respective assignments. This approach contrasts with the independent, service-oriented structure of MSA, where

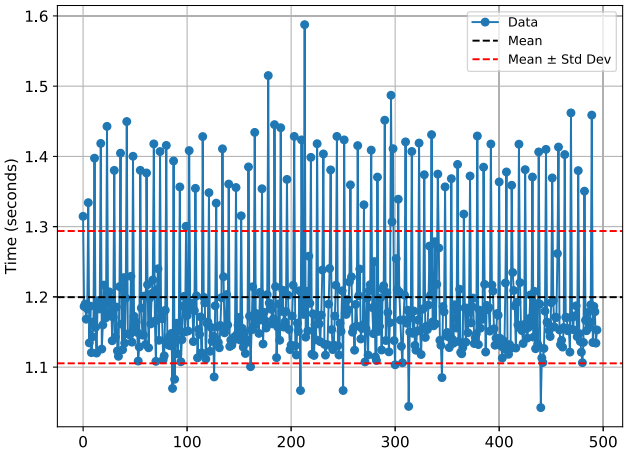


FIGURE 14. MSA - Processing time with mean and standard deviation.

each service operates autonomously and communicates over well-defined APIs.

Figure 13 and Figure 14 present the standard deviation of processing times for MA and MSA, respectively. While both architectures exhibit similar overall variability, the MSA distribution appears more tightly clustered, indicating greater consistency in processing times. One possible explanation for the higher dispersion observed in MA is that all tasks are executed on the same processing unit, leading to increased I/O contention and fluctuations in response time due to resource competition.

In summary, the analysis indicates that MA performs better in processing and delivery times under normal conditions, while MSA demonstrates superior performance during high-load scenarios. Experiments show that MA suffers from increased hardware consumption due to resource contention, while MSA alleviates this by distributing requests across individual machines, improving overall performance. Furthermore, processing and delivery times revealed consistent averages, with slight variations for a single user. These findings highlight the effect of architectural choices on the overall system performance depending on specific operational conditions.

V. RELATED WORK

This section reviews selected related works involving Microservice Architecture (MSA), with a focus on systems implemented using Machine Learning (ML), Natural Language Processing (NLP), and other supporting technologies. The objective is to identify relevant similar contributions in the literature and extract insights that inform and contextualize the present study. Table 3 summarizes the related works discussed throughout this section.

A. NLP IN MSA

Some studies employed NLP techniques to facilitate interaction with end users, either through the retrieval and

classification of information or the identification of microservices that align with and support business objectives.

Bhagat and Robins [24] proposed a solution aimed at ensuring data security while granting data scientists access to the necessary information, without requiring the distribution of the core system software. Their approach integrates data from distributed microservices to support diverse analytical tasks. Specifically, three microservices were developed, each implemented in a different programming language and equipped with its own dedicated database. One service stores customer feedback related to specific products or services; another captures customer-related metadata, such as login history, timestamps, and exchanged messages; and the third aggregates all collected data and applies NLP techniques to classify user comments into sentiment categories, negative, neutral, or positive, for future analysis. To validate the approach, the authors conducted a sentiment analysis of user feedback from e-commerce platforms.

Brito et al. [25] introduced a novel methodology for identifying microservices through topic modeling, organizing them based on the domain-specific features of the original monolithic software. This approach was implemented as an open-source tool designed to analyze MA and support their transformation into MSA. Leveraging NLP techniques, the tool identifies potential microservices directly from the MA source code. The methodology comprises three main steps: Information Extraction, Topic Modeling, and Clustering. By extracting both structural and lexical data from the source code, it facilitates a more effective migration to MSA. The evaluation was carried out using a GitHub dataset of 200 open-source Spring Boot applications, and the results demonstrated strong effectiveness, particularly in maintaining functional independence between the identified services.

B. ML IN MSA

Some studies explored the nuances of implementing ML approaches as microservices. For instance, modularizing ML components not only accelerates deployment and simplifies maintenance but also enables systematic evaluation of application performance to ensure scalability, reliability, and efficient resource utilization.

Pahl and Loipfinger [26] encapsulated ML as a data-driven microservice tool designed to integrate applications within ML-specific contexts. They analyzed three distinct types of ML algorithms: Feed-Forward Neural Networks, Deep Belief Networks, and Recurrent Neural Networks. They applied their approach in an Internet of Things (IoT) scenario, aiming to decouple the algorithm implementation from its configuration. The experiments demonstrated that modularizing ML applications can significantly accelerate their deployment as ML services. Although latency is not a critical concern in many IoT use cases, the authors suggested mitigation strategies such as caching mechanisms. However, latency remains a significant factor that could render the model unfeasible in other scenarios.

Ribeiro et al. [27] proposed a general-purpose architecture known as Machine Learning in Microservice Architecture (MLMA) to support the development of ML pipelines by modularizing similar processing steps into distinct microservices. This architecture introduces design patterns that facilitate the transformation of a monolithic ML pipeline into a microservice-based system, with clearly defined roles for each service. The MLMA framework is adaptable to a wide range of projects involving data classification tasks. It is structured around a five-step workflow comprising the Controller, Processing, Collector, Orchestration, and Classification components. The architecture was evaluated through two case studies within a Smart City initiative, demonstrating significant improvements in maintainability and code reusability across different applications.

Andyartha et al. [28] proposed an evaluation framework that integrates ISO/IEC 25023:2016 metrics with several key activities in the testing life cycle of a cloud-based ML in MSA. This framework, based on [29], equips practitioners with guidelines that consist of metrics and threshold selection for effective evaluation. The biggest challenge for an ML application is ensuring optimal performance within a heterogeneous environment. The model was distributed across four different microservices: Identify Testing, Adjust Thresholds, Execute Test, and Evaluate Results. The experiments were carried out in a healthcare system, where the model assists doctors in diagnosing patients based on patient information and interpreting medical images. The reliability evaluation of the model was carried out with patient data as input to identify patterns. Promising results showed high reliability while confirming that the proposed framework enhanced the system's reliability characteristics.

Kaushik et al. [34] developed a model where a micro frontend has been used in the design of the user interface, and microservices have been used in the design of the backend architecture. An empirical analysis was carried out to investigate the performance of the proposed model and compare it with an existing monolithic frontend following a microservices architecture-based application. A regression ML model was built to predict the response time of MSA-based applications, which can be used for making performance improvement decisions. The results show that the model achieved an accuracy of 66.75%, and the limited availability of datasets for microservice-based applications contributed to the relatively low accuracy. Nonetheless, this performance can potentially be improved by applying optimization and other techniques.

C. MA VS MSA

Some studies compared MA and MSA, highlighting the benefits and drawbacks of migration. These studies evaluate system performance and scalability under various scenarios and test configurations to better understand the behavioral differences between the two approaches.

Ma et al. [23] introduced an information retrieval approach called Scenario-based MicroService Retrieval (SMSR). The

model aims to provide a retrieval algorithm service based on the Word2vec algorithm, which is widely used in NLP, to help with the selection and similarity analysis of services. The method proposed in this work to identify microservices consists of five steps: (i) extract the relevant noun and verb terms; (ii) process these terms using a Word2Vec model; (iii) filter the candidate services based on specific linguistic criteria; (iv) calculate the semantic similarity between the terms; and (v) rank the qualified microservices and return the top result to the user. Experiments showed that SMSR retrieves information faster and more efficiently than a monolithic application setup.

Kleftakis et al. [30] compared the performance of the BeHealtheir platform in both MA and MSA. This platform is capable of handling heterogeneous health data for management and analysis using several ML algorithms. To validate the experiments, the platform was deployed across three different types of microservices in a cluster with Docker Compose and microservices orchestrated within Kubernetes clusters. Approximately a dozen ML algorithms were employed, including Naive Bayes, Decision Tree, and Logistic Regression, among others. The results showed that the platform deployed using the MSA outperformed the same platform hosted in the MA. This performance gain was particularly evident with larger datasets, although it introduced greater complexity in the model training process. Nevertheless, only minor variations were observed across different MSA deployment configurations.

Al-Debagy and Martinek [31] conducted a comparative analysis of MA and MSA in terms of performance, aiming to evaluate their behavior under various scenarios and test configurations. The proposed solution comprises three components: the first component connects all other components within the MSA; the second handles all back-end communication tasks; and the third serves as the front-end interface. JMeter was used in the experiments to measure the performance of the solution in both architectural environments across two test scenarios. The results indicated that in the load test, both architectures performed similarly, with a slight performance advantage observed in the MSA. However, in the concurrency test, the MA outperformed the MSA by approximately 6% in processing time when the number of concurrent users reached around 100.

Reimer [32] explored orchestration and choreography within a given semantic transformation scenario. To validate the comparison between choreography and orchestration, the System Development Research Methodology (SDRM) was the solution used, which is a sequential approach to evaluate research results in the final phase of an observation. In the context of this work, NLP was used to formulate semantic transformation queries within the scope of the back-end graph structure, which also provides an additional layer of abstraction for the front-end request system. Experiments were performed for both types of workflows for a case of semantic transformation that provides both benefits and drawbacks. As a result, the choreography-based approach showed a high degree of adaptation to certain environments, while orchestration showed better performance in the analysis of the structure of sentences when executed in the backend.

Gos and Zabierowski [33] conducted a comparative analysis of MA and MSA to assess their performance and scalability, using a Java-based application developed with Spring Boot. The system was organized into three layers: Web Layer, Service Layer, and Repository Layer. The database was implemented using PostgreSQL, while Docker was used to containerize the services. The MSA implementation was a variation of the monolithic application, implemented using Spring Boot along with Spring Cloud. Performance testing was carried out using Gatling, with both POST and GET requests applied to each architecture. The findings revealed that while both architectures presented strengths and weaknesses in specific scenarios, the MSA outperformed the MA under high-load conditions, where the MSA evaluation was conducted using a single database instance.

Ramamoorthi [35] explored how AI techniques can be integrated into microservice architectures to improve both performance and resource efficiency. It proposed an AI-driven optimization framework that combines Reinforcement Learning (RL), Predictive Analytics (PA), and Evolutionary Algorithms (EA) to dynamically manage resources in distributed, highly scalable microservices environments. The results indicate that the AI-based optimization system offers notable enhancements in both performance and resource efficiency within microservice architectures, outperforming traditional approaches. Decreased latency and increased throughput also demonstrate the system's ability to manage dynamic workloads more efficiently. The integration of RL and PA for real-time decision-making and workload forecasting was crucial for sustaining high performance, especially during peak demand periods such as traffic spikes. This study presents a novel NLP application designed with a MSA to enable precise performance evaluation. It compares monolithic and microservice designs in terms of processing and response times, introduces orchestration for coordinated service execution, and includes a dedicated, non-intrusive statistics service to support system optimization and knowledge analysis.

Upon analyzing Table 3, we conclude that this work is unique because it compares MA and MSA architectures specifically within NLP applications, an area often overlooked by other research. Unlike studies that generally examine architecture migration or performance across various domains, this research analyzes processing times at different stages of an NLP pipeline, whether microservices are used or not. Additionally, it provides insights into microservices' operational efficiency, scalability, and performance tracking through an orchestrator service and a dedicated SG module, all without affecting the main application's throughput. The study's focus on real-time message processing in a C2 context and its analytical framework further enhances the discussion

TABLE 3. List of related works.

Reference	Year	Objective	Domain	MA	MSA	ML	NLP
Ma *et al.* [23]	2018	Model for identifying microservices based on the scenario approach	Microservice Identification		✓		✓
Pahl and Loipfinger [26]	2018	Implement a machine learning application as a microservice	Information Extraction in IoT		✓	✓	
Al-Debagy and Martinek [31]	2018	Monolithic and Microservice architecture comparison	Performance Evaluation Architectures	✓	✓		
Ribeiro *et al.* [27]	2019	Architecture to migration (Monolithic to Microservice)	Smart City to Business	✓	✓	✓	
Bhagat *et al.* [24]	2020	Customer sentiment analysis on microservice	E-commerce Services and Products		✓		✓
Gos and Zabierowski [33]	2020	Comparison of microservice and monolithic	E-commerce Business Logic	✓	✓		
Brito *et al.* [25]	2021	Identifying microservices through topic modeling	Open-source Spring Boot Applications		✓		✓
Kleftakis *et al.* [30]	2022	Comparison Docker and Kubernetes applied to MSA	Healthcare Applications		✓	✓	
Reimer [32]	2023	Orchestration and choreographic approach	Semantic Transformations		✓	✓	
Andyartha *et al.* [28]	2023	Reliability evaluation of ML application on MSA	Software Quality		✓	✓	
Kaushik *et al.* [34]	2024	Front-End Performance: MSA and ML	Application Design		✓	✓	
Ramamoorthi *et al.* [35]	2024	Enhanced Performance Optimization for Microservice	Microservice Optimization	✓	✓		

on architectural decisions for domains needing continuous improvement.

VI. CONCLUSION

MSA appeared as an alternative to traditional monolithic systems, enabling application decomposition into smaller, independent services. This approach delivers significant benefits in terms of scalability and fault isolation, especially in distributed environments. By evaluating its performance, it is possible to consider the trade-off between processing speed under normal conditions and its ability to handle high loads. While MSA may introduce complexities in deployment, its ability to distribute resource demand across services allows for improved performance in highly concurrent scenarios.

In this paper, we discuss the performance of an NLP application for exchanging textual messages in a C2 environment, comparing the performance of the MSA and MA versions. The main contribution of this application is to provide a comparison between these two architectures for C2 applications, specifically for a message prioritization and enhancement application. After evaluating the performance results of all scenarios, including both processing and delivery times, it can be concluded that MA's performance was essentially better than that of MSA. This improvement, however, was limited to a few milliseconds for processing time and a few seconds for delivery time. Conversely, when a large volume of messages is handled simultaneously, MSA exhibited better delivery times compared to MA. This improvement is likely more related to network performance rather than the MAISC^2 system's engine itself. In a monolithic setup, all applications are hosted on a single machine, and all traffic passes through a single network card, whereas this is not the case in a microservices architecture. Consequently, if an application serves a limited number of users and does not require extensive scalability, MA can be a suitable choice. However, for applications with a large user base and future scalability needs, MSA is the better choice.

Several challenges emerged during the implementation process. For instance, service orchestration requires careful coordination of multiple independent services to ensure smooth message exchange. This was mitigated by introducing an orchestrator service to handle dependencies effectively. Another significant challenge was data performance. Unlike in MA, where all components share a single database, MSA relies on distributed data management. To address this, different database implementations, such as SQLite and PostgreSQL, were selected based on each service's specific requirements. In this way, performance was prioritized over consistency.

Therefore, a key insight of our experimental findings is the identification of specific tradeoffs for NLP-based C2 systems when implemented under different architectural styles. While previous studies in other domains have shown that MSA is beneficial for scalability and fault isolation, our results highlight that under normal load conditions, MA still performs better in terms of processing and delivery times.

However, MSA demonstrated superior performance in high concurrency scenarios, which is particularly relevant in C2 contexts where sudden communication spikes may occur.

Some limitations were identified in the work that reduced the possibilities of broader experiments and the range of observing metrics. One notable constraint derives from the military domain itself, where access to historical data is highly restricted. As a result, a smaller dataset had to be used to train the model, which influenced the development of the algorithms responsible for message prioritization. Another limitation lies in the scenario for evaluating the model in a microservices environment. A simplified use case was employed, involving message exchanges among three users, one acting as the command and the other two as subordinate units, thereby limiting the system's evaluation under more complex and dynamic operational conditions.

Another limitation and potential topic for further work was the decision not to optimize process efficiency by placing all containers on a single physical machine, which could have minimized external interferences such as network latency or hardware variability. While this approach might have yielded more controlled results, the intention was to simulate a more realistic deployment scenario, especially for C2 environments where services are commonly distributed across multiple nodes. This configuration enabled the observation of how MSA behaves under real-world conditions, while also recognizing the trade-offs involved in adopting this setup without the benefit of complementary evaluations in more controlled environments.

For future work, we plan to optimize MSA performance by implementing asynchronous task queues, such as Celery, to increase scalability and enable more efficient parallel processing. Additionally, incorporating load-balancing techniques and optimizing algorithms may improve resource distribution and reduce processing bottlenecks.

Unlike general MSA applications, C2 systems often involve real-time, high-stakes communication where message prioritization and semantic clarity should be supported. This introduces specific requirements for low-latency processing and consistent delivery, even under stress or peak loads. One challenge we observed was orchestrating semantically complex tasks across distributed services without introducing significant delays, which is often not present in conventional MSA applications. Additionally, domain-specific entity recognition (e.g., military units, operations, and directives) requires datasets to be built as well as the customization of NLP components. Due to the private nature of the real data, the training task may be hindered and time-consuming. Furthermore, in the case of joint C2 operations, different institutions interoperate (e.g., civil defense, navy, air force, etc.), and their vocabulary often differs. Consequently, different versions of the NLP components have to be trained with specific datasets and deployed accordingly to the respective users within a single operation.

Moreover, while reliability is an important metric in evaluating this type of system, it was considered outside the scope of this study due to its specific focus on performance-related aspects. A different experimental setup is under consideration to assess failure recovery. Furthermore, the MAISC^2 MSA approach has the potential to be applied to other applications, possibly in other domains that require term classification in message exchanges. Nonetheless, definitive conclusions on the general applicability of this approach and its implementations can only be made after conducting more experiments.

ACKNOWLEDGMENT

The authors would like to thank João Luiz Rebelo Moreira from the University of Twente and Luís André Gomes de Abreu from the Military Institute of Engineering, who supported the development team on the use of the infrastructure for the experiments.

REFERENCES

[1] K. Gao, H. Wang, J. Nazarko, and M. Jarocka, "Adaptive decision method in C3I system," *Comput. Intell. Neurosci.*, vol. 2022, pp. 1–17, Aug. 2022, doi: 10.1155/2022/6967223.

[2] J. Simpson, R. Oosthuizen, S. E. Sawah, and H. Abbass, "Agile, antifragile, artificial-intelligence-enabled, command and control," 2021, *arxiv:2109.0687*.

[3] F. F. Da Silva Mosafi, L. F. Pires, J. C. Duarte, and M. C. Cavalcanti, "Towards a microservices architecture to support communication in C2 applications," in *Proc. 19th Annu. Syst. Syst. Eng. Conf. (SoSE)*, Tacoma, WA, USA, Jun. 2024, pp. 227–232, doi: 10.1109/SOSE62659.2024.10620941.

[4] E. Hosseini, A. M. Al-Ghaili, D. Hussein Kadir, F. Daneshfar, S. Shamini Gunasekaran, and M. Deveci, "The evolutionary convergent algorithm: A guiding path of neural network advancement," *IEEE Access*, vol. 12, pp. 127440–127459, 2024, doi: 10.1109/ACCESS.2024.3452511.

[5] N. Jayatilleke, R. Weerasinghe, and N. Senanayake, "Advancements in natural language processing for automatic text summarization," in *Proc. 4th Int. Conf. Comput. Syst. (ICCS)*, Sep. 2024, pp. 74–84, doi: 10.1109/ICCS62594.2024.10795848.

[6] W. Qu and J. Li, "Research on named entity recognition algorithm based on NLP," in *Proc. IEEE 12th Int. Conf. Inf., Commun. Netw. (ICICN)*, Aug. 2024, pp. 559–565, doi: 10.1109/ICICN62625.2024.10761899.

[7] P. Sun, X. Yang, X. Zhao, and Z. Wang, "An overview of named entity recognition," in *Proc. Int. Conf. Asian Lang. Process. (IALP)*, Bandung, Indonesia, Nov. 2018, pp. 273–278, doi: 10.1109/IALP.2018.8629225.

[8] T. Wang, H. Li, K. Xiao, and H. Huang, "Field named entity recognition based on double layer conditional random fields," in *Proc. 9th Int. Conf. Big Data Inf. Analytics (BigDIA)*, Dec. 2023, pp. 640–645, doi: 10.1109/BIGDIA60676.2023.10429568.

[9] D. Premarathna and A. Pathirana, "Theoretical framework to address the challenges in microservice architecture," in *Proc. Int. Res. Conf. Smart Comput. Syst. Eng. (SCSE)*, vol. 4, Sep. 2021, pp. 195–202, doi: 10.1109/SCSE53661.2021.9568346.

[10] A. Krylovskiy, M. Jahn, and E. Patti, "Designing a smart city Internet of Things platform with microservice architecture," in *Proc. 3rd Int. Conf. Future Internet Things Cloud*, Aug. 2015, pp. 25–30, doi: 10.1109/FICLOUD.2015.55.

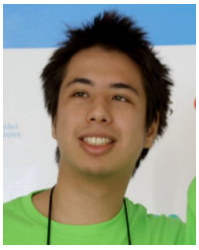
[11] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables DevOps: Migration to a cloud-native architecture," in *Proc. 33.3 IEEE Softw.*, vol. 33, Mar. 2016, pp. 42–52, doi: 10.1109/MS.2016.64.

[12] D. Kang, K. J. Seo, and T. Kim, "Revisiting the impact of pursuing modularity for code generation," 2024, *arXiv:2407.11406*.

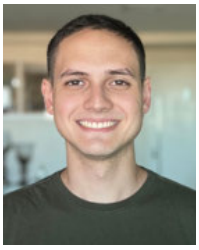
[13] A. Kapikul, D. Savić, M. Milić, and I. Antović, "Application development from monolithic to microservice architecture," in *Proc. 28th Int. Conf. Inf. Technol. (IT)*, Zabljak, Montenegro, Feb. 2024, pp. 1–4, doi: 10.1109/it61232.2024.10475769.

[14] H. Knoche and W. Hasselbring, "Using microservices for legacy software modernization," *IEEE Softw.*, vol. 35, no. 3, pp. 44–49, May 2018, doi: 10.1109/MS.2018.2141035.
[15] D. Wolfart, W. K. G. Assunção, I. F. da Silva, D. C. P. Domingos, E. Schmeing, G. L. D. Villaca, and D. D. N. Paza, "Modernizing legacy systems with microservices: A roadmap," in *Proc. 25th Int. Conf. Eval. Assessment Softw. Eng.*, New York, NY, USA, Jun. 2021, pp. 149–159, doi: 10.1145/3463274.3463334.
[16] A. Haque, R. Rahman, and S. Rahman, "Microservice-based architecture of a software as a service (SaaS) building energy management platform," in *Proc. 6th IEEE Int. Energy Conf. (ENERGYCon)*, Gammarth, Tunisia, Sep. 2020, pp. 967–972, doi: 10.1109/ENERGYCON48941.2020.9236617.
[17] F. Ferreira, J. Duarte, and W. Ugulino, "Automated statistics extraction of public security events reported through microtexts on social networks," in *Proc. 18th Brazilian Symp. Inf. Syst.*, May 2022, pp. 1–7, doi: 10.1145/3535511.3535513.
[18] X. Wang, R. Yang, Y. Lu, and Q. Wu, "Military named entity recognition method based on deep learning," in *Proc. 5th IEEE Int. Conf. Cloud Comput. Intell. Syst. (CCIS)*, Nanjing, China, Nov. 2018, pp. 479–483, doi: 10.1109/CCIS.2018.8691316.
[19] R. Chen, S. Li, and Z. Li, "From monolith to microservices: A dataflow-driven approach," in *Proc. 24th Asia–Pacific Softw. Eng. Conf. (APSEC)*, Dec. 2017, pp. 466–475, doi: 10.1109/APSEC.2017.53.
[20] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," in *Proc. Present ulterior Softw. Eng.*, 2017, pp. 195–216.
[21] G. Blinowski, A. Ojdowska, and A. Przybylek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," *IEEE Access*, vol. 10, pp. 20357–20374, 2022, doi: 10.1109/ACCESS.2022.3152803.
[22] C. A. J. Acevedo, J. P. Gómez y Jorge, and I. R. Patiño, "Methodology to transform a monolithic software into a microservice architecture," in *Proc. 6th Int. Conf. Softw. Process Improvement (CIMPS)*, Zacatecas, Mexico, Oct. 2017, pp. 1–6, doi: 10.1109/CIMPS.2017.8169955.
[23] S.-P. Ma, Y. Chuang, C.-W. Lan, H.-M. Chen, C.-Y. Huang, and C.-Y. Li, "Scenario-based microservice retrieval using Word2Vec," in *Proc. IEEE 15th Int. Conf. e-Business Eng. (ICEBE)*, Oct. 2018, pp. 239–244, doi: 10.1109/ICEBE.2018.00046.
[24] V. Bhagat, "Natural language processing on diverse data layers through microservice architecture," in *Proc. IEEE Int. Conf. Innov. Technol. (INOCON)*, Nov. 2020, pp. 1–6, doi: 10.1109/INOCON50539.2020.9298027.
[25] M. Brito, J. Cunha, and J. Saraiva, "Identification of microservices from monolithic applications through topic modelling," in *Proc. 36th Annu. ACM Symp. Appl. Comput.*, New York, NY, USA, Mar. 2021, pp. 1409–1418, doi: 10.1145/3412841.3442016.
[26] M.-O. Pahl and M. Loipfinger, "Machine learning as a reusable microservice," in *Proc. IEEE/IFIP Netw. Oper. Manage. Symp.*, Apr. 2018, pp. 1–7, doi: 10.1109/NOMS.2018.8406165.
[27] J. L. Ribeiro, M. Figueredo, A. Araujo, N. Cacho, and F. Lopes, "A microservice based architecture topology for machine learning deployment," in *Proc. IEEE Int. Smart Cities Conf. (ISC2)*, Oct. 2019, pp. 426–431, doi: 10.1109/ISC246665.2019.9071708.
[28] P. K. Andyartha, U. L. Yuhana, A. B. Raharjo, and D. Purwitasari, "Presenting a reliability evaluation framework for cloud-based machine learning in microservices," in *Proc. 6th Int. Seminar Res. Inf. Technol. Intell. Syst. (ISRITI)*, Dec. 2023, pp. 95–100, doi: 10.1109/ISRITI60336.2023.10467653.
[29] S. Karnouskos, R. Sinha, P. Leitão, L. Ribeiro, and T. I. Strasser, "The applicability of ISO/IEC 25023 measures to the integration of agents and automation systems," in *Proc. 44th Annu. Conf. IEEE Ind. Electron. Soc.*, Oct. 2018, pp. 2927–2934, doi: 10.1109/IECON.2018.8592777.
[30] S. Kleftakis, A. Mavrogiorgou, N. Zafeiropoulos, K. Mavrogiorgos, A. Kiourtis, and D. Kyriazis, "A comparative study of monolithic and microservices architectures in machine learning scenarios," in *Proc. IEEE Int. Conf. Comput. (ICOCO)*, Nov. 2022, pp. 352–357, doi: 10.1109/ICOCO56118.2022.10031648.
[31] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," in *Proc. IEEE 18th Int. Symp. Comput. Intell. Informat. (CINTI)*, Nov. 2018, pp. 149–154, doi: 10.1109/CINTI.2018.8928192.
[32] T. Reimer, "Architecture to productize semantic transformation by utilizing natural language processing with microservices," in *Proc. Int. Conf. Adv. Enterprise Inf. Syst. (AEIS)*, Dec. 2023, pp. 35–41, doi: 10.1109/aeis61544.2023.00013.
[33] K. Gos and W. Zabierowski, "The comparison of microservice and monolithic architecture," in *Proc. IEEE 14th Int. Conf. Perspective Technol. Methods MEMS Design (MEMSTECH)*, Apr. 2020, pp. 150–153, doi: 10.1109/MEMSTECH49584.2020.9109514.
[34] N. Kaushik, H. Kumar, and V. Raj, "Micro frontend based performance improvement and prediction for microservices using machine learning," *J. Grid Comput.*, vol. 22, no. 2, p. 44, Jun. 2024, doi: 10.1007/s10723-024-09760-8.
[35] V. Ramamoorthi, "AI-enhanced performance optimization for microservice-based systems," *J. Adv. Comput. Syst.*, vol. 4, no. 9, pp. 1–7, 2024, doi: 10.69987/jacs.2024.40901.
[36] K. Sellami and M. Aymen Saied, "Contrastive learning-enhanced large language models for monolith-to-microservice decomposition," 2025, *arXiv:2502.04604*.
[37] F. Daneshfar, S. Soleymanbaigi, A. Nafisi, and P. Yamini, "Elastic deep autoencoder for text embedding clustering by an improved graph regularization," *Expert Syst. Appl.*, vol. 238, Mar. 2024, Art. no. 121780, doi: 10.1016/j.eswa.2023.121780.

FLAVIO FERREIRA DA SILVA MOSAFI received the M.Sc. degree in computer science from the Military Institute of Engineering (IME), Rio de Janeiro, Brazil, in 2021, where he is currently pursuing the Ph.D. degree in computer science. His research interests include large language models, natural language processing, and microservices architecture. Apart from the research, he is engaged in activities with software development. Besides this, he is actively involved in machine learning, deep learning, artificial intelligence, semantic interoperability, ontology, and command and control research.



MATEUS SZE COSENZA is currently pursuing the degree in computer engineering with the Military Institute of Engineering (IME). He has participated in ICPC contests and has participated in the world championship twice. He enjoys problem-solving and also teaches upcoming ICPC participants at his university advanced algorithms and data structures.



LÉO VICTOR CRUZ VASCONCELOS is currently pursuing the degree in computer engineering with the Military Institute of Engineering (IME). He works with IT infrastructure, cloud computing, and Linux, constantly exploring new technologies and best practices in the field. His passion for computing spans a wide range of topics from digital electronics to the mathematical foundations of algorithms. Beyond his professional work, he is deeply interested in understanding and optimizing computational systems at all levels, always seeking to bridge the gap between theory and practice.

LUÍS FERREIRA PIRES (Member, IEEE) received the B.Sc. degree in electronics from the Instituto Tecnológico de Aeronáutica, São José dos Campos, Brazil, in 1983, the M.Sc. degree in electrical engineering from the Escola Politécnica da Universidade de São Paulo, São Paulo, Brazil, in 1989, with a focus on the computer networks, and the Ph.D. degree from the Faculty of Electrical Engineering, University of Twente, Enschede, The Netherlands, in September 1994. Since 1988, he has been at the University of Twente, initially as a Research Fellow at the Faculty of Computer Science, from 1992 to 1994, as an Assistant Professor at the Faculty of Electrical Engineering, and has been an Associate Professor at the Faculty of Electrical Engineering, Mathematics and Computer Science, since 1994. His research interests include design methodologies for distributed systems, the architecture of distributed systems, modeling and specification techniques, middleware platforms, and telematics applications.

JULIO CESAR DUARTE received the Graduate degree from the Military Institute of Engineering (IME), in 1998, and the master's and Ph.D. degrees in computer science from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), in 2003 and 2009, respectively. He complemented his education by completing a postdoctoral internship at PUC-Rio, in 2021. He is currently a Professor of the postgraduate program in systems and computing and the Pro-Rector of teaching and research at IME. He is a professional with academic training and experience in computer engineering. His academic and professional performance is marked by a multidisciplinary approach, with a significant emphasis on developing computer systems. His experience spans several areas of computing, including artificial intelligence, machine learning, deep learning, and Portuguese natural language processing. In addition, he has been conducting research on multimodal media processing, large-scale language models, and malware analysis.

MARIA CLAUDIA REIS CAVALCANTI received the D.Sc. degree in systems and computer engineering from the Federal University of Rio de Janeiro, Brazil, in 2003. She is currently a Full Professor with the Department of Computer Engineering, Military Institute of Engineering (IME), Rio de Janeiro, Brazil. Previously, she was a Systems Analyst at the Federal University of Rio de Janeiro (UFRJ), from 1985 to 2004. Since 2004, she has been a Lecturer at IME. She has also worked as a Researcher and an Advisor of the Postgraduate Program in Systems and Computer Science (PGSC) and the Postgraduate Program in Defense Engineering (PGED). Her current research interests include metadata and ontologies for data interlinking, conceptual modeling, semantic web (of data), data modeling for NoSQL DBMS, and, more recently, cyber-security and command and control (C2) systems. In addition, she has coordinated and participated in research projects on those topics, with funding from Brazilian government agencies, including CNPq, CAPES, FINEP, and FAPERJ. In 2000, she received the CNPq Scholarship for a postgraduate visit to INRIA-Rocquencourt, France, while pursuing her Ph.D. studies.

Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - ROR identifier: 00x0ma614