Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 238 (2024) 224–231

Procedia Computer Science

www.elsevier.com/locate/procedia

The 15th International Conference on Ambient Systems, Networks and Technologies (ANT)
April 23-25, 2024, Hasselt, Belgium

A Distributed Software Architecture for IoT: Container Orchestration Impact and Evaluation

Gustavo M. Freire[a,*], Herminio Paucar Curasma[a], Julio Cezar Estrella[a]

[a]*Institute of Mathematical and Computer Sciences (ICMC), Av. Trabalhador São Carlense, São Carlos, Brazil*

Abstract

This paper proposes a Distributed Software Architecture (DSA) for Smart Building (SB) based on the Reactive Manifesto (RM) principles. To follow the RM principles, we analyze the usage of different deployment approaches, particularly the impact of using a container orchestrator on the application layer. After running performance tests on the different configurations, the container orchestrator usage led to enhanced distributed processing, lowering the latency, increasing flexibility, enhancing security, and providing cost-effectiveness and scalability. We introduce the implementation of a modern DSA, developed following state-of-the-art cloud patterns and compliant with the RM for the SB context. Furthermore, we have ensured the reproducibility of this implementation by making the initial tests and overall architecture code available in public repositories. The research follows the Design Science Research (DSR) methodology for elaborating each phase until we get the artifact (DSA) and, with this, contribute to the Knowledge Base. The architecture was properly tested, considering the performance as the principal test layer. This solution is tailored for application in domains of the Internet of Things (IoT), focusing on the SB and a case study involving the Laboratory of Distributed Systems and Concurrent Programming (LaSDPC) at São Paulo University. Moreover, its applicability extends to IoT domains like smart home, smart campus, smart city, and health-related applications.

Keywords: Distributed architecture; reactive manifesto; cloud computing; microservices; IoT

1. Introduction

The IoT is a disruptive technology that transforms buildings into SB. IoT is a growing area of research, as evidenced by the increasing number of publications on the topic [17]. DSA is necessary for implementing modern IoT solutions because it improves scalability, making adding more sensors and actuators easy without compromising performance. It improves the reduction of latency and increases the process distribution capacity. A DSA helps reinforce security because many nodes make it difficult for intruders to act [18]. This paper aims to propose and test the initial nodes of

* Corresponding author. Tel.: +55 22 98835-0003
E-mail address: gustavofreire@usp.br

10.1016/j.procs.2024.06.019

a DSA that follows the principles of RM[1], cloud patterns, and cloud computing capable of supporting the complex and scalable demands of a system for an SB that will be applied in the LaSDPC, and this project is part of a big project that includes Multi-Agent System, Ontologies and Federated Learning (see[2] for the complete project). The main contributions of this paper are as follows.

- We introduce the implementation of a modern DSA, developed following state-of-the-art cloud patterns and compliant with the RM for the SB context. Furthermore, we have ensured the reproducibility of this implementation by making the initial tests and overall architecture code available in public repositories [3] [4] .
- This architecture should offer the flexibility to be implemented in public or private clouds with similar implementation costs, thus being agnostic to the target infrastructure, contributing to critical data protection (private cloud deployments) and novelty approaches to IoT (edge computing deployments).
- High adaptability to specific and different use cases while maintaining low operational costs and delivering exceptional performance is also desired.
- We aim to contribute with a system guided by the most recent best practices in distributed software engineering for IoT applications.
- The performance of the proposed system is evaluated using K6, an open-source tool for concurrent load tests.

Thus, we raise the following research question *Would it be possible to develop a DSA following the reactive principles, being agnostic to different models of cloud computing and capable of being deployed on novelty edge computing devices ?*. To answer this question, we want to make a two-phase process. The first phase aims to simulate scenarios using synthetic data and perform tests where public and private clouds will be used. The second phase aims to deploy the system in the LaSDPC private cloud to use physical sensor data. It is then intended to propose a cloud-based agnostic system that deals with different use cases through a base architectural model. This paper covers the initial first-phase performance tests and analysis on a small-scale but representative portion of the architecture (the client/read nodes).

2. Related Works

We searched the most relevant works in popular libraries (IEEExplorer, ACM DL, Web of Science, and Scopus). Afterward, the selected articles (see Table 2) were critically analyzed, considering the comparison and evaluation criteria (Table 1). These criteria were obtained from other studies[8, 16] that compare architectural characteristics, highlighting the type of technologies, techniques, or paradigms. Unlike most works analyzed, our proposed architecture and paper introduce use cases and benchmarking and provide the source code necessary for reproducibility. The work also introduces, in a broader sense, the usage of Reactivity by managing the system with a container orchestrator.

3. Architecture proposal

This research aims to create a versatile SB architecture using cloud computing, emphasizing the application and network layers in IoT systems (IEEE2413 standard and IoT-A [2]). The main contribution is a distributed architecture based on reactive principles, adaptable to diverse data scenarios, cloud environments, and network topologies. It leverages cloud-native patterns for elasticity, resilience, monitoring, and cost efficiency. By using a container orchestrator, this architecture presents the capability of being deployed on edge devices, thus contributing to the state-of-art IoT use cases.

Figure 1 displays the the proposed architecture; following, we describe each element and their responsibilities:

[1] https://www.reactivemanifesto.org/
[2] https://smart-lasdpc.github.io/about-project.html
[3] https://github.com/Smart-LaSDPC/Distributed-Software-Architecture
[4] https://github.com/Smart-LaSDPC/reactive-architecture-smart-building

Table 1. Criteria for analyzing related work

Id	Criteria	Considerations
C01	Architecture model	Microservices, service-oriented architecture (SOA), monolithic, or other models were considered.
C02	Communication model	Communication between services, whether asynchronous or synchronous, was considered.
C03	Deployment Model	Private, public, community, or hybrid models were considered.
C04	Network layer	Edge-of-the-network (EoT), fog-of-the-network (FoT), and cloud-of-the-network (CoT) layers were considered.
C05	Deployment technology	Technologies such as containers, serverless computing, orchestrators, and choreography were considered.
C06	Design patterns	The use of software patterns, as well as patterns specific to the cloud, were considered.
C07	Observability	The observability of the system, whether through metrics, logging, or tracing, was considered.
C08	Reactivity	The principles outlined in the Reactive Manifesto in the design of services and system nodes] were considered.
C09	Handling Error	Error handling, such as reprocessing and feedback mechanisms, was considered.
C10	Use Cases	the implementation of use cases was considered.
C11	Benchmarking	Benchmarking and comparisons with other types of architectures and frameworks were considered.
C12	Reproducibility	The references, data, and information necessary to reproduce the results presented were considered.

Table 2. Related works evaluated in each criterion

Cri.	**Stud.1**[10]	**Stud.2**[7]	**Stud.3**[4]	**Stud.4**[3]	**Stud.5**[14]	**Stud.6**[6]	**Stud.7**[15]	**Stud.8**[13]	**Stud.9**[11]	**Stud.10**[5]
C01	MSA	MSA	MSA	MSA, SOA	MSA	MSA, SOA	MSA, SOA	MSA	MSA	MSA
C02	ASYNC, Event-driven, Pub/Sub	SYNC, REST	SYNC, REST, RPC	X	ASYNC, SYNC	SYNC Http/ Rest, AMQP ASYNC, Event-driven, Pub/Sub	SYNC Http/ Rest, ASYNC AMQP, Message-Oriented, Pub/Sub	SYNC, Http/ Rest	SYNC, Http/ Rest	ASYNC, Message-passing
C03	X	X	X	X	X	X	X	X	X	X
C04	IoT	FoT	IoT	CoT, FoT, EoT	CoT, FoT, EoT	CoT, FoT, EoT	FoT, EoT	FoT, CoT	EoT, FoT, CoT	X
C05	X	X	Orches., Choreo., Contai.	X	Orches., Service Mesh, Contai.	Orches., Choreo., Contai.	X	Orches., Service Mesh, Contai.	X	VMs, OpenStack
C06	Yes	No	Yes	No	Yes	Yes	Yes	No	No	Yes
C07	No	No	Yes	No	Yes	Yes	No	Yes	No	Yes
C08	Yes	No	No	Yes	Yes	Yes	Yes	No	Yes	Yes
C09	Yes	No	Yes	No	No	Yes	Yes	No	Yes	No
C10	No	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes
C11	No	Yes	No	No	No	No	Yes	No	Yes	Yes
C12	No	No	No	No	No	No	No	No	No	Yes

- **Messenger/Brokers:** Store events generated by sensors and provide them to consumers using the publisher/subscriber model.
- **Database:** Efficiently store and manage data, accommodating structured or unstructured information with relational or non-relational characteristics.
- **Cache:** Store frequently requested data more efficiently, typically using hash mapping data structure.
- **API Gateway:** Manage access points for all services, overseeing both external and internal access, security, and monitoring mechanisms.
- **Ingestors:** Consume data from broker topics, responsible for preprocessing, transforming, validating, and projecting data to other data stores.
- **Circuit Breaker:** Oversee the lifecycle of topics and the retry pattern, serving as a synchronization mechanism for data streams and error handling.
- **Heartbeat:** Send events to topics to keep consumers (or consumer groups) active, preventing timeouts caused by inactivity.
- **Rest/GRPC API:** Provide a REST and GRPC interface to various clients and between microservices.
- **Websocket API:** Offer an interface to diverse clients, enabling persistent and bidirectional connections, facilitating real-time data transmission.

Tools descriptions: For the selection of technologies, considerations encompassed the required licensing types, community size and support, efficiency and scalability, the feasibility of employing distributed strategies, resource concurrency, and synergy with reactive, microservices-oriented, and cloud-native systems.

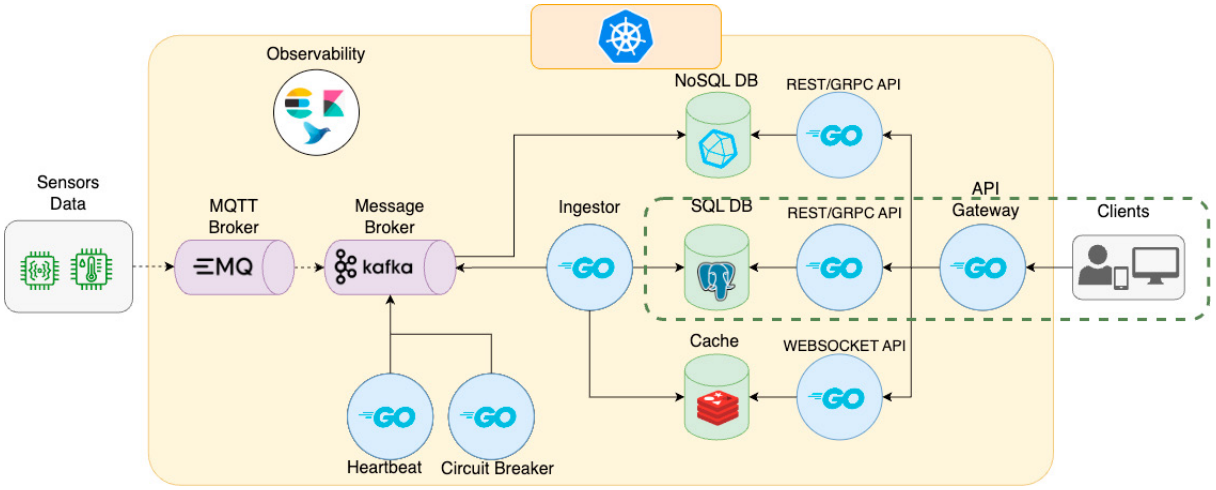

Fig. 1. Architecture with proposed technologies. The green dotted square highlights the nodes tested in this article.

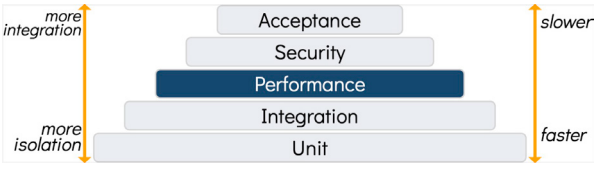

Fig. 2. Test layer selected for the proposed architecture

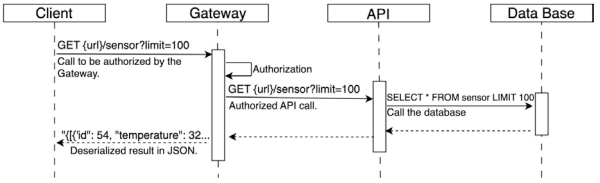

Fig. 3. Sequence diagram of the Orchestrator Experiment

4. Tests and Evaluations

Testing holds a pivotal role within the software development lifecycle. According to [9], we can separate tests into five layers, as displayed in Figure 2. The *Performance* level was selected for this work. The performance test plays an integral role in verifying the software's capability to withstand anticipated loads across diverse scenarios, thereby shedding light on scalability and system expansion potential. The subsequent sections outline the methodology employed for conducting the testing assessments.

- **Identification of the key performance indicators (KPIs):** For this work, the metrics selected were Response Time, Throughput, Network Traffic, and Error Rates as they are standard metrics used for performance evaluation [1].
- **Definition of the Load profiles:** The Load profiles were based on two different statistical distributions, the Uniform and Poisson. The load profiles also defined different request volumes, including 50, 500, 5000, and 50000 concurrent requests.
- **Selection of loading testing tools:** The open-source K6 tool, which was developed specifically for automated performance and load testing of APIs and Microservices, was selected. K6 provides valuable metrics by default. Among the non-functional metrics, the following stand out: quantity of data sent, quantity of data received, number of blocked requests, duration of requests, number of failures, requests per second, and quantity of data sent.

  K6 provides statistical information for all of its metrics, such as average value, minimum, maximum, and two sigmas of standard deviations. It also offers the possibility of custom load customization with different types of statistical distributions. For tests on messaging and databases, custom scripts will also be used to generate the aforementioned metrics.

- **Test Tool Configuration and Execution:** The tools were configured using *Javascript* scripts, The tests scheduler was configured using bash scripts and cronjobs services.
- **Analyze of the test results:** The results were analyzed and described through diagrams and tables.

Table 3. Comparison of percentages.

TID	Experiment A				Experiment B				Experiment C			
	1	2	3	4	1	2	3	4	1	2	3	4
Concu. Clients	50	500	5000	50000	50	500	5000	50000	50	500	5000	50000
Dur.(fcst/curr)	30s/ 30s	30s/ 34.6s	30s/ 60s	30s/ 60s	30s/ 31.1s	30s/ 34.7s	30s/ 60s	30s/ 60s	30s/ 30.7s	30s/ 33.3s	30s/ 60s	30s/ 60s
Requests Qty	1438	4178	7126	19549	1382	2077	6372	50563	1450	4993	9270	27068
Requests/sec	46.35/s	120.81/s	118.75/s	322.13/s	44.48/s	59.85/s	106.19/s	875.57/s	47.29/s	149.81/s	154.48/s	603.74/s
Recv data	263MB	764MB	1.3GB	980MB	253MB	380MB	423MB	381MB	265MB	913MB	1.7GB	1.7GB
	8.5MB/s	22MB/s	22MB/s	16MB/s	8.1MB/s	11MB/s	7.1MB/s	6.3MB/s	8.7MB/s	27MB/s	28MB/s	28MB/s
%Faults	0%	0 %	0 %	72.59%	0%	0%	63.70%	96.04%	0%	0%	0%	74.43%
Requests Dur.												
avg	61.19ms	2.85s	26.09s	7.59ms	104.34ms	6.87s	4.73s	450.16ms	49.16ms	2.15s	22.98s	7.73s
min	16.78ms	87.99ms	63.38ms	0s	50.46ms	264.9ms	0s	0s	17.42ms	75.82ms	158.34ms	0s
max	422.29ms	6.85s	57.55s	58.65s	816.79ms	15s	43.97s	59.38s	443.94ms	6.96s	59.02s	59.38s
Block Dur.												
avg	14.56μs	2.04ms	51.4ms	315.79ms	601.9μs	49.2ms	37.66ms	31.74ms	20.72μs	2.19ms	58.39ms	205.42ms
min	1.21μs	1.02μs	1.17μs	0	795.μs	942ns	0s	0s	1.47μs	1.26μs	1.02μs	0s
max	538.94μs	74.89ms	349.97ms	58.34s	21.08ms	5.49μs	15.44s	20.25s	1.06ms	114.85ms	429.68ms	4.27s

5. Experiments

One of the prominent challenges encountered in the domains of distributed systems and IoT lies in comprehending the implications on system scalability and efficiency, particularly given the growing volume and sources of data, often heterogeneous, integrating with the system. Therefore, being able to observe how the system reacts to different data inputs, with different frequencies and structures, is a crucial aspect of our research.

The aspect of the proposed architecture tested in this phase, the orchestrator, forms the basis for the hypothesis of this research and is therefore essential for elucidating and validating previous choices. The container orchestrator contributes to the system's reactivity by providing resilience, elasticity, responsiveness, and also contributes to greater scalability considering the same computational resources. The use of cloud computing was also evaluated, with experiments executed both locally and on a public cloud.

All resources and code used have been shared in a public git repository with the aim of enabling result reproducibility.

Planning: For our Orchestrator experiments, we set up a database with synthetic data based on real sensors, a Gateway microservice, and a backend microservice responsible for communicating with the database. These components are essential for our system's scalability considerations [12], as they are commonly used in various scenarios where data needs to pass through security, ingestion for processing, and storage.

Our experiment focuses on testing the system's performance under different configurations by applying various levels of workload. In all configurations, we use the same test flow, ensuring that requests are consistent and return the same data (a query that retrieves 100 sensor data records). Figure 3 shows the sequence diagram of an experiment using the orchestrator.

The activities that we follow for this reasearch are i) Data modeling, (ii) Environment setup, iii) Testing tools profiles execution, iv) Individual analysis of the experiments, and v) Comparative analysis of the experiments.

Data Modeling: To conduct our experiments, we structured a data model based on attributes from real temperature and air conditioning sensors. The data was modeled using a relational database (PostgreSQL) and the types indicated to follow: The table includes critical attributes represented by temperature_current (F), temperature_desired (F), humidity (F), pressure (F), air_quality (F), voltage (F), current (F), power (F), is_on (B), status (V), location (V), and extracted_at (T), denoting float, boolean, varchar, and timestamp data types, respectively. Moreover, the 'status' field delineates specific operational modes such as 'actuating,' 'ventilator,' 'heater,' 'cooling,' and 'off'.

These data were populated into the database through a SQL seed script, with parameters aligned with the real-world range of the synthesized quantity.

We developed a data structure to map the database model and convert (serialize) it to JSON using the Go programming language. Additionally, endpoints were exposed to insert and retrieve data through a Go-written REST API.

Environment Setup: Below (Figure 4), we present the infrastructure configuration and the resources utilized on all performed experiments (A, B and C).

Experiment A: we implemented the load testing tool, the gateway microservice, and the backend microservice in a locally configured environment, equivalent to a private cloud. The database was set up in a public cloud (AWS) using the RDS service. The cloud-based database service was chosen for its advantageous features, including ease of configuration, high availability, scalability, elasticity, security, and monitoring. Hence, the environment for the first experiment can be considered a hybrid cloud setup.

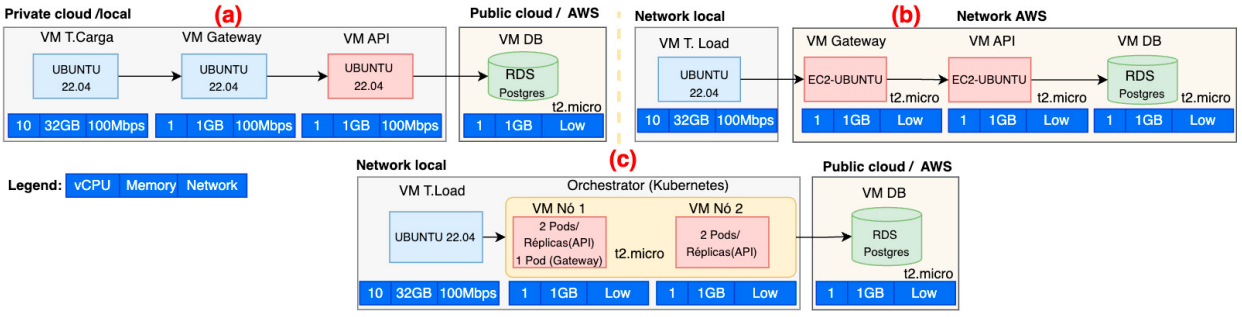

Fig. 4. Resources and configurations for the experiments (a) A, (b) B and (c) C.

Experiment B: only the load testing tool was implemented in the local network, while the microservices and the database were configured in the public cloud (AWS) using EC2 and RDS services. This environment, excluding the load testing tool hosted locally, constitutes a native public cloud setting.

Experiment C: we implemented all microservices in a local networked environment, with the database residing in the public cloud (AWS) using the RDS service. Unlike the first experiment, we employed a container orchestrator for microservice management. The orchestrator was configured to utilize two nodes (two VMs) with three Pods, where the Pods responsible for backend containers hosted two replicas each.

Testing Tools Profiles Execution: For test execution, we employed the K6 tool to run various profiles of concurrent request loads on the system, simulating diverse scenarios and demands. Both tests incorporated four distinct load types: 50 concurrent requests for 30 seconds, 500 concurrent requests for 30 seconds, 5000 concurrent requests for 30 seconds, and 50000 concurrent requests for 10 minutes.

Performance evaluation utilized response variables, including response time, request error rate, transactions per second, and throughput.

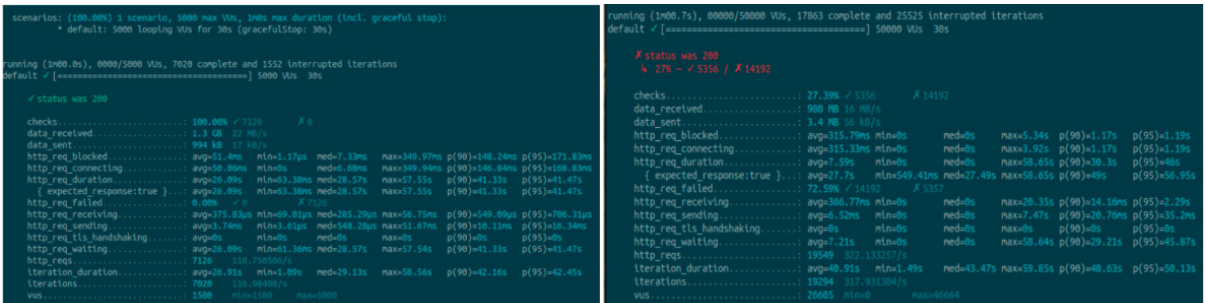

Fig. 5. Sample of a Testing Tool Profile Metrics Outputs

Individual analysis of the experiments: We describe the three experiments (A, B, and C) in Table 3.

Experiment A: presents the results generated during all iterations of the load test. Notably, the data received for test 3 (5000 concurrent requests) amounts to 1.3GB, slightly exceeding that of test 2 (764MB). This can be attributed to the higher number of requests processed over a longer duration in test 3 (7126 in 60s) compared to test 2 (4178 in 34.6s). Furthermore, it is noteworthy that test 4 (50000 concurrent requests) exhibits a substantial failure rate of 72.59%, indicating the concurrency threshold for this architecture with the available resources.

Experiment B: presents the results obtained from all iterations of the load test. Notably, there is a high failure rate in test 3 (5000 concurrent requests) at 63.70% and in test 4 (50000 concurrent requests) at 96.04%. These results indicate the concurrency threshold for this architecture with the provided resources. Additionally, it is important to highlight that the low average duration times for these tests (3 and 4) signify system failure and are not indicative of performance or throughput.

Experiment C: presents the results, with particular attention to the data received and the number of requests per second. In test 3 (5000 concurrent requests), data received amounts to 1.7GB, with 154,486 requests per second. It is worth mentioning the significant failure rate in test 4 (50000 concurrent requests) at 74.43%, marking the concurrency limit for this configuration.

Comparative analysis of the experiments: Based on the results obtained in the previous section, we can now proceed to compare the outcomes of the experiments. In Figure 6 presented below, these insights are graphically represented.

Fig. 6. Comparative Analysis of Experiments

In the first graph (Data Received) and the third one (Data Received Rate), we can observe Experiment 3 holding a distinct advantage over the other experiments in all tests. This highlights that the use of the orchestrator and replicas has significantly enhanced data reception capacity and system throughput. This phenomenon can be attributed to the use of replicas, where the orchestrator itself employs the Round Robin load balancing algorithm, reducing system idleness during request blocking, thereby increasing parallelism and scalability. This improvement is particularly evident in the fourth graph (Average Duration of Blocks), where this experiment delivers favorable figures, especially in test 2 (500 concurrent iterations).

Furthermore, it is evident from the graphs that Experiment B yields the poorest results across all criteria. This can be attributed to the system being hosted in a public cloud, resulting in latency overhead during data transfer between the public cloud and the local environment. This latency issue is clearly illustrated in the second graph (Failure Percentage), where Experiment B exhibits a significantly higher failure percentage, even at 5000 concurrent requests, in contrast to the other experiments.

6. Conclusions and Future works

This paper presents our initial DSA proposal and performance test results. We proved experimentally that using a container orchestrator positively impacts the reactiveness and scalability of the solution. We also concluded that using a private cloud decreases the latency, especially compared to the hybrid configuration. The technology stack provided cost-effectiveness and reproducibility by utilizing an open-source technology stack. With this preliminary version, our contributions to the state-of-the-art include (i) the use of a distributed architecture for the IoT context following the RM; (ii) the applications of cloud patterns such as API Gateway and container orchestration to implement an architecture that can be deployed on different hosts, such as edge devices; (iii) the creation of a solution that will allow

its architecture to be applied to other IoT applications in different sectors (smart home, smart hospital, smart campus, industry 4.0, and others);

In future works, we will extend the tests to cover the other nodes of the architecture. We will also integrate the distributed architecture with its complementary components, such as Multi-Agent systems, Federated Learning, Ontologies, and a context-aware approach. We aim to showcase the combined functionality in a real-world scenario, precisely the LaSDPC, where we will rigorously test each principle of the RM. LaSDPC provides an ideal infrastructure for researching Distributed Systems due to its extensive array of devices, including sensors and actuators, and a data center comprising five racks that support diverse research projects.

Acknowledgements

This work was developed using the computational infrastructure of the Distributed Computing Lab of ICMC-USP—the University of São Paulo (http://infra.lasdpc.icmc.usp.br/) and also with resources from the Center for Mathematical Sciences Applied to Industry (CeMEAI http://www.cemeai.icmc.usp.br/) funded by the São Paulo Research Foundation FAPESP (Project Number 2013/07375-0) and also by Project Number 11/09524-7).

References

[1] Badawy, M., El-Aziz, A.A., Idress, A.M., Hefny, H., Hossam, S., 2016. A survey on exploring key performance indicators. Future Computing and Informatics Journal 1, 47–52. doi:10.1016/j.fcij.2016.04.001.

[2] Bauer, M., Boussard, M., Bui, N., Carrez, F., (SIEMENS, C., (ALUBE, J., (SAP, C., Meissner, S., IML, A., Olivereau, A., (SAP, M., Joachim, W., Stefa, J., Salinas, A., 2013. Internet of Things – Architecture IoT-A Deliverable D1.5 – Final architectural reference model for the IoT v3.0. URL: https://www.iot-a.eu/public/front-page/.

[3] Óscar Belmonte-Fernández, Sansano-Sansano, E., Trilles, S., Caballer-Miedes, A., 2022. A reactive architectural proposal for fog/edge computing in the internet of things paradigm with application in deep learning doi:10.1007/978-3-030-84459-2_9.

[4] Butzin, B., Golatowski, F., Timmermann, D., 2016. Microservices approach for the internet of things. 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA) doi:10.1109/ETFA37052.2016.

[5] Debski, A., Szczepanik, B., Malawski, M., Spahr, S., Muthig, D., 2018. A scalable, reactive architecture for cloud applications. IEEE Software 35, 62–71. doi:10.1109/MS.2017.265095722.

[6] Dobaj, J., Krisper, M., Iber, J., Kreiner, C., 2018. A microservice architecture for the industrial internet-of-things. EuroPLoP '18: Proceedings of the 23rd European Conference on Pattern Languages of Programs doi:10.1145/3282308.3282320.

[7] Khoso, F.H., Arain, A.A., Nizamani, S.Z., Lakhan, A., Soomro, M.A., Kanwar, K., 2021. A microservice-based system for industrial internet of things in fog-cloud assisted network. Engineering, Technology and Applied Science Research (ETASR) doi:10.1002/spe.2729.

[8] Konersmann, M., Kaplan, A., Kühn, T., Heinrich, R., Koziolek, A., Reussner, R., Jürjens, J., al Doori, M., Boltz, N., Ehl, M., Fuchs, D., Groser, K., Hahner, S., Keim, J., Lohr, M., Sağlam, T., Schulz, S., Töberg, J.P., 2022. Evaluation methods and replicability of software architecture research objects, in: 2022 IEEE 19th International Conference on Software Architecture (ICSA), pp. 157–168. doi:10.1109/ICSA53651.2022.00023.

[9] Medhat, N., Moussa, S., Badr, N., Tolba, M.F., 2019. Testing techniques in iot-based systems, in: 2019 Ninth International Conference on Intelligent Computing and Information Systems (ICICIS), pp. 394–401. doi:10.1109/ICICIS46948.2019.9014711.

[10] Molina, J., M., Garcia, J., F., Jimenez, C., K., 2018. Archer: An Event-Driven Architecture for Cyber-Physical Systems. IEEE/ACM International Conference on Utility and Cloud Computin Companion , 335–340.

[11] Power, A., Kotonya, G., 2018. A microservices architecture for reactive and proactive fault tolerance in iot systems. 2018 IEEE 19th International Symposium on "A World of Wireless, Mobile and Multimedia Networks (WoWMoM) doi:10.1109/WoWMoM.2018.8449789.

[12] Richards, M., 2015. Software Architecture Patterns. 3 ed., O'Reilly Media.

[13] Salah, T., Zemerly, M.J., Yeun, C.Y., Al-Qutayri, M., Al-Hammadi, Y., 2018. Iot applications: From mobile agents to microservices architecture doi:10.1109/INNOVATIONS.2018.8605967.

[14] Sanctis, M.D., Muccini, H., Vaidhyanathan, K., 2020. Data-driven adaptation in microservice-based iot architectures. 2020 IEEE International Conference on Software Architecture Companion (ICSA-C) doi:10.1109/ICSA-C50368.2020.00019.

[15] Santana, C., Andrade, L., Delicato, F.C., Prazeres, C., 2021. Increasing the availability of iot applications with reactive microservices. Service Oriented Computing and Applications - Volume 15 - Issue 2 doi:10.1007/s11761-020-00308-8.

[16] Simmhan, Y., Ravindra, P., Chaturvedi, S., Hegde, M., Ballamajalu, R., 2018. Towards a data-driven iot software architecture for smart city utilities. Software: Practice and Experience 48, 1390–1416. doi:10.1002/spe.2580.

[17] Suzuki, L.R., 2017. Smart Cities IoT: Enablers and Technology Road Map. Springer International Publishing, Cham. pp. 167–190. doi:10.1007/978-3-319-61313-0_10.

[18] Tanenbaum, A., van Steen, M., 2017. Distributed Systems. CreateSpace Independent Publishing Platform.