Towards Applying API Gateway to support Microservice Architectures for Embedded Systems

Miroslav Tomić*, Vladimir Dimitrieski*, Marko Vještica*, Radovan Župunski*, Aleksandar Jeremić*, Hannes Kaufmann**

*University of Novi Sad, Faculty of Technical Sciences, 21000 Novi Sad, Serbia

** KEBA AG, 4010 Linz, Austria

{tmiroslav, dimitrieski, marko.vjestica, radovan.zupunski, jeremicaleksandar}@uns.ac.rs, kauf@keba.com

***Abstract*—A development of contemporary industrial production systems is heavily influenced by the fourth industrial revolution. These systems are undergoing a shift from mass production to mass customization. To support mass customization, Industrial Control Systems (ICSs) should be implemented in a modular and flexible way. One way to achieve this goal is to apply a microservice architecture in the context of ICSs. Since such an architecture has not been frequently applied in ICSs recently, it is still considered as an open research challenge. The adoption of microservices in ICSs increases the number of Application Programming Interfaces (APIs) which could be burdensome both for developers who maintain API integrations and users who need to be aware of too many service endpoints. Therefore, a microservice architecture should include an API gateway module to alleviate these issues and provide a single point of connectivity. In this paper, we identify and describe the requirements for an API gateway solution to be applied in the domain of ICSs. These requirements are then used to identify one or more API gateways that are the most suitable to be applied in embedded systems .**

I. Introduction

Most of the currently used industrial production systems are built to support mass production. They belong to the era of Industry 3.0 (I3.0) in which industrial systems are mostly centralized, monolithic, and complex. Nowadays, a customer demands specific and customized products, implying frequent changes in production. These frequent changes cause a shift from mass production to mass customization. This shift towards the mass customization has caused what is now known as the fourth industrial revolution or Industry 4.0 (I4.0), for short. Mass customization requires more flexible engineering solutions than currently used Industrial Control Systems (ICSs) in I3.0. ICSs are used as a general term to encompass several types of control systems such as Supervisory Control And Data Acquisition (SCADA) systems, Distributed Control Systems (DCS), and Programmable Logic Controllers (PLCs). Since ICSs are often implemented in a monolithic way, they need to be converted into modular, distributed, collaborative, and product-service oriented systems to address frequent changes as required by I4.0 [1].

To support decentralization of ICSs, the microservice architecture, or microservices for short, may be applied. Microservices represent an architectural style that emphasizes loosely coupled modules allowing easy scaling, deployment, and independent development [2]. Also, microservices provide a basis for the development of systems that are flexible, highly maintainable, and business-oriented [3]. Such a decentralized architecture may also contribute to the creation of modular and distributed ICSs.

The transition toward a microservice architecture implies an increase in the number of independent services as well as the number of Application Programming Interfaces (APIs). The increase in the number of APIs is burdensome for developers since they need to maintain those APIs. Additionally, users of ICSs often need to be aware of too many services and create complex integrations to fully utilize ICSs. Over time, new services may be added, or existing services may be further instantiated to scale up a system, which requires even more architecture awareness from a user. Also, service addresses and ports may change dynamically and various services may use different protocols to communicate. Therefore, to overcome this nonuniformity and an increase in the number of APIs, an API gateway may be used as a single entry point into a system [3]. As an API gateway is linked with services, users may communicate with services through the gateway without knowing all the different service addresses and port numbers. Additionally, an API gateway can perform different tasks such as API composition, authentication, authorization, and load balancing.

In recent years, microservices have been applied in the domain of embedded systems, which are a part of ICSs. Unlike applications in common domains such as e-commerce, telecommunication, health care systems, and finance, in the domain of embedded systems there are additional limitations and requirements that have delayed an application of a microservice architecture. On the one hand, some limitations are a direct consequence of limited resources or available runtime environments for running microservices. On the other hand, a usage of a real-time communication protocol and timing constraints are of higher importance in the domain of embedded systems than in the aforementioned domains. All these limitations and requirements must be taken into consideration when choosing an API gateway to be used

in an embedded microservice architecture as it is commonly run at the same system or device as other ICS microservices.

In order to contribute to solving the aforementioned challenges, the main goal of our research is to integrate a microservice architecture with an API gateway into embedded systems, considering a resource consumption and facilitating a real-time communication [4]. The aim of this paper is to identify and provide a description of requirements for an API gateway module which is to be used in such a microservice architecture and compare multiple API gateway solutions that are in line with those requirements.

Apart from Introduction and Conclusion, this paper is organized as follows. In Section 2, an overview of the related work is given. Research methodology is described in Section 3, while identified requirements for applying an API gateway into embedded systems are addressed in Section 4. An evaluation and a comparison of API gateway solutions are presented in Section 5.

II. Related Work

During the literature review, we encountered only a few research papers related to the application of an API gateway in embedded systems. Therefore, we investigated what are limitations for the adoption of a microservice architecture in the domain of embedded systems. Also, we tried to identify requirements for using an API gateway in a microservice architecture for embedded systems. Besides the reviewed literature, we have searched for existing solutions and industrial use cases as well.

Several authors have discussed different approaches to implement a microservice architecture in the industrial automation domain, as well as challenges that emerged with its adoption. Homay et al. [1] examined the advantages and disadvantages of microservices over a service-oriented architecture in the context of industrial automation. A migration path, starting with an ICS monolith codebase and going towards a microservice architecture has been suggested by Buchgeher et al. [4]. The authors have also provided a systematic overview of principles related to a microservice architecture, highlighting their relation to steps of the proposed migration path. However, none of the mentioned papers contains any implementation details of microservice architectures, such as the service discovery and load balancing.

A more technical analysis of the microservices application in embedded systems is presented by Wang et al. [5]. The authors proposed a framework that would aid an implementation of microservices in embedded systems. The specification of this framework includes a service proxy component in charge of data format conversions and requests handling. This service proxy component, albeit being similar to an API gateway, provides only a small subset of functional requirements that is expected of an API gateway solution.

Mathijssen et al. [6] provided an overview of the API management terminology and also establish an API gateway as a design pattern in relation to the topic of the API management. Zhao et al. [7] discussed an application of API gateway in microservice architectures. The authors discussed functionalities expected to be performed by an API gateway in a microservice-based system, and they also described technical implementation details regarding the architecture of a single API gateway solution such as an API gateway authentication, and API Gateway Reverse Proxy Function. Further discussion regarding the implementation details of API gateway solutions can be found in [8] and [9]. However, these papers do not examine the issues specific to the API gateway implementation in the context of embedded systems such as a need for a real-time communication or imperative for a low resource usage.

A usage of an API-gateway-supported microservice solution in embedded systems has been documented by Xu et al. [10], too. The authors discussed an implementation of a security agent in an edge computing platform. The agent described in this paper is based on an open-source API gateway solution named *Kong Gateway*[a], but the authors did not provide a rationale behind choosing the Kong API gateway over other available solutions.

During the literature review we did not find any paper providing an overview of requirements an API gateway solution should satisfy to be used in embedded systems. Also, we did not find an overview of existing API gateway solutions and a comparison of their resource usage on embedded devices. Accordingly, in this paper we present a description of requirements, an overview of existing API gateway solutions and a comparison of their resource usage on embedded devices.

III. Research Methodology

To address the main goal of our research, we follow the Design Science Research Methodology (DSRM) [11]. Usually, DSRM includes six iterative steps: (i) problem identification and motivation; (ii) defining the objectives for a solution; (iii) design and development; (iv) demonstration; (v) evaluation; and (vi) communication.

During the literature review we identified a problem of applying an API gateway into a microservice architecture for embedded systems. Also, we identified requirements that came from both the literature and the industry needs as a part of the DSRM first step. As an outcome of the first step, we obtained a better overview of the current state of a microservice architecture in the domain of embedded systems. This helped us to define objectives of our research as a part of the DSRM second step. Our objectives are to: (i) identify and describe requirements for applying an API gateway solution in embedded systems; and (ii) integrate a solution that satisfies those requirements in a microservice architecture. Therefore, we searched for API gateway solutions that may be integrated in embedded systems, as a part of the DSRM third step. We have identified a plethora of API gateway solutions that may fit our needs. First, we made an initial selection of API gateways by reading their documentation and investigating whether they fulfill identified requirements. Afterwards, to demonstrate a usage of selected solutions, we applied them in a small-scale industrial setup with embedded devices, as a part of the DSRM fourth step. As the outcome of this application, API gateway solutions were eliminated if they could not

[a] https://konghq.com/kong

fulfill some of the mandatory requirements, described in the following section. In the DSRM fifth step, we collected data from embedded devices of our setup and evaluated the performance of remained API gateways. As a part of the DSRM sixth step, we aim to publish our findings in this paper.

IV. API GATEWAY RESEARCH RESULTS

The results of this research are presented through the following subsection. In the first subsection, requirements we have encountered are presented in a structured form. The following subsection presents API gateway solutions we have found during our research. The final subsection provides a description of our industrial setup and a usage of API gateway solutions in it.

A. Identified Requirements

After reviewing the current state in the domain of embedded systems and API gateways, we have determined that there is an industrial need for an application of an API gateway solution in a microservice architecture for embedded systems. Also, requirements for applying an API gateway solution in such systems are coming from both the literature and the industry. We have discussed about manufacturing and programming of industrial PLCs with domain experts working at a medium-sized Original Equipment Manufacturer (OEM). Thus, we have gathered useful knowledge about the adoption of a microservice architecture in the domain of embedded systems.

A structured overview of the identified mandatory and optional requirements for an API gateway to be used in embedded systems are presented in the following lists.

Mandatory requirements are:

- **Standalone and Open-Source Solution (OSS)**. An API gateway should be a standalone and OSS that is regularly maintained. This requirement is preferred by the industry as companies would usually like to adapt the solution for their needs.
- **Advanced RISC Machines (ARM), x86, and amd64 processor architectures**. These processor architectures should be supported, as they are most often used in embedded devices.
- **Linux and Windows Operating Systems (OSs).** Since embedded devices commonly use these two OSs, an API gateway should be runnable on them.
- **Linux and Windows installer packages.** It should be possible to make installer packages from an API gateway solution, such as Debian (deb) and Microsoft installer (msi) packages, providing an easy deployment of the API gateway.
- **HyperText Transfer Protocol (HTTP), HyperText Transfer Protocol Secure (HTTPS), and WebSocket (WS) communication protocols.** The most frequently used protocols for microservices and therefore must be supported.
- **REpresentational State Transfer (REST) architectural style.** This is the basic and most common architectural style for the implementation of microservices and therefore must be supported.
- **Secure Sockets Layer (SSL) protocol.** To enable a secure communication between embedded devices, the SSL protocol should be supported by an API gateway.
- **Authentication.** An API gateway should communicate with the authentication service, thus eliminating a need for other services to be connected with it.
- **Small flash usage.** An API gateway should use as small amount of flash memory as possible on embedded devices. Domain experts that participated in this research stated that the flash usage should be smaller than 200MB.
- **Small memory footprint.** An API gateway should have small memory footprint when being in the idle state. Domain experts stated that the memory footprint should be smaller than 100MB on embedded devices.
- **Low Control Processing Unit (CPU) usage.** A CPU usage of an API gateway should be as low as possible on embedded devices, as devices usually have weak hardware.

Optional requirements are:

- **Server-Sent Events (SSE) communication protocol.** The SSE protocol has been used for a real-time communication and as it is preferred for embedded devices, it would be beneficial for an API gateway to support it.
- **Load balancing.** To enhance the scalability of microservices, it would be beneficial that an API gateway support it.
- **Dynamic reconfiguration.** To ensure higher flexibility, a dynamic reconfiguration of an API gateway allows to reconfigure it without a need to stop embedded devices and thus the production can continue without losses.
- **Standalone webserver.** Since many services need to be deployed on an embedded device, it would be beneficial that an API gateway is a standalone webserver to enable such the deployment.

B. API Gateway Solutions

To find different API gateway solutions, Google Search Engine was used, and a plethora of API gateway solutions were identified. We recorded 31 solutions as potential candidates. An initial selection of those solutions was made by checking their documentation and finding out whether they are open-source and deployable in a standalone manner. As a result of the initial selection, 19 solutions remained for the further analysis: *Kong Gateway*, *Tyk*[b], *Apache APISIX*[c], *pushpin*[d], *janus*[e], *KrakenD*[f], *Lura*[g], *Ambassador*[h], *WSO2 API Microgateway*[i], *Goku*[j], *Express Gateway*[k], *Easegress*[l],

[b] https://tyk.io/
[c] https://apisix.apache.org/
[d] https://pushpin.org/
[e] https://hellofresh.gitbooks.io/janus/content/
[f] https://www.krakend.io/
[g] https://luraproject.org/
[h] https://www.getambassador.io/
[i] https://mg.docs.wso2.com/en/latest/
[j] https://www.gokuapi.com/
[k] https://www.express-gateway.io/
[l] https://megaease.com/easegress/

spring-cloud-gateway[m], *Zuul 2*[n], *WSO2*[o], *Ocelot*[p], *Gravitee*[q], *adobe-apiplatform apigateway*[r] and *Fusio*[s].

After the initial selection of API gateways, we read the documentation further to check other requirements fulfilment and to determine solutions that will be tested on embedded devices. In this phase, we eliminated *Fusio*, *Ocelot*, *WSO2*, *Zuul 2*, *adobe-apiplatform apigateway*, and *Gravitee* from further analysis. The decision was made as these solutions could not be run on the ARM architecture or there was no information about it in the documentation.

In Table I, we present API gateway solutions and their fulfilment of the requirements. Each table cell has one of the following values:

- **Y** – a requirement is fulfilled.
- **N** – a requirement is not fulfilled.
- **P** – a requirement is partially fulfilled, meaning that an API gateway supports a functionality, but a plug-in needs to be added or the functionality needs to be manually implemented.
- **N/A** – a requirement fulfilment information could not be determined based on the documentation.

Based on the requirements fulfilment presented in Table I, we decided to test the following API gateway solutions on embedded devices: *Kong Gateway*, *Tyk*, *Apache APISIX*, *pushpin*, *Ambassador* and *WSO2 API Microgateway*.

C. An Industrial Setup with Embedded Devices

To test selected API gateway solutions, we prepared a small-scale industrial setup, consisting of one low-end device and two high-end devices. Most frequently used embedded devices are usually equipped with relatively weak hardware and such devices are called low-end devices. On the opposite, embedded devices equipped with stronger hardware are called high-end devices. We have used the following embedded devices to test the API gateway solutions:

- **Device 1 (high-end device)** – debian-based Linux OS, amd64 processor architecture, Intel i7 core, 4GB of Random Access Memory (RAM) and 15GB of flash memory;
- **Device 2 (low-end device)** – debian-based Linux, armhf processor architecture, ARM Cortex-A9, 512MB of RAM and 2GB of flash memory; and
- **Device 3 (high-end device)** – Windows OS, i386 processor architecture, Intel Celeron, 4GB of RAM and 32GB of flash memory.

On these devices, we failed to setup *Ambassador* and *WSO2 API Microgateway* for the basic usage. These API gateways solutions were not compatible with the OS versions running on the devices. Therefore, we decided to eliminate these solutions for the further analysis. We managed to setup *pushpin* on *Device 1* and *Device 2*, but failed to setup it on *Device 3*, as *pushpin* does not support Windows OS. There is a possibility to build a *pushpin* source code for Windows, but we could not manage it in a reasonable amount of time. *Tyk* does not officially support Windows as well, but we were able to build its source code for Windows. Both *pushpin* and *Tyk* API gateways, as well as two other remaining API gateways, *Kong Gateway* and *Apache APISIX*, have been tested on the three embedded devices of our industrial setup. A performance evaluation of these API gateways is described in the following section.

[m] https://spring.io/projects/spring-cloud-gateway

[n] https://github.com/Netflix/zuul

[o] https://wso2.com/api-manager/

[p] https://ocelot.readthedocs.io/en/latest/introduction/gettingstarted.html

[q] https://www.gravitee.io/

[r] https://github.com/adobe-apiplatform/apigateway

[s] https://www.fusio-project.org/

V. Performance Evaluation of API Gateways on Embedded Devices

To obtain data about a flash usage on *Device 1* and *Device 2*, we have used the Linux command disk free (df) and to obtain data on *Device 3*, we have used the Windows OS system utility disk management. To measure an API gateway solution flash usage on a device, we subtracted the flash usage before and after the API gateway has been installed. In Table II we present a flash usage of API gateways on *Device 1*. The *Usage (MB)* column represents an API gateway flash usage expressed in Megabytes (MB). It should be noted that *Kong Gateway* and *Apache APISIX* failed to fulfill the flash usage requirement, using more than 200MB of flash memory. Therefore, we did not test these two API gateways further.

The remaining API gateways were tested on embedded devices by measuring the CPU and RAM usage in three different test scenarios. In the first test scenario, HTTP requests were sent to API gateways, while in the second test scenario, WS requests were sent to them. In the third test scenario, both HTTP and WS requests were sent in parallel to API gateways. Each test scenario contained sub-scenarios in which a request sequence of 1, 6, 10, 20 and 50 requests were sent in parallel. Every test scenario was run twice – with and without the SSL protocol enabled. A request sequence was repeated three times to ensure that measurements were similar each time. After a request sequence is finished, there was an idle time of 5 seconds, allowing an API gateway to finish any residue processing. The test scenarios were implemented in Python programming language.

To measure a resource usage on *Device 1* and *Device 3*, the Linux OS Top utility was utilized, showing the summary information of the CPU and RAM usage, grouped by active processes. Each value obtained from the Top utility represents a percentage of a resource usage. The Top utility also provides a possibility to measure a resource usage on the desired time interval. In our test scenario, the CPU and RAM were measured every 100 milliseconds. Since *Device 2* had the Windows OS, a custom PowerShell script was written, providing the same information as the Top utility.

Values given in Tables III through V represent measurements gathered on devices from the Top utility and the PowerShell script. These values represent an API gateway resource usage expressed in percentages. Slashes in Table IV and Table V represent cases when we were unable to run some test scenarios, which is discussed further in this section. Table columns are organized as follows: (i) *No SSL* column represent data gathered when the script was run without the SSL protocol enabled; (ii)

TABLE I. REQUIREMENT MATRIX

API Gateway	Requirements																			
	OSS	HTTP	HTTPS	REST	SSL	X86	amd64	Linux	Authentication	deb	msi	ARM	Windows	WS	Flash	RAM	CPU	SSE	Load Balancing	Dynamic Reconfiguration	Standalone Webserver
Kong Gateway	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	P	Y	N	N/A	N/A	N/A	Y	Y	Y
Tyk	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N/A
Apache APISIX	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	P	Y	N	N/A	N/A	N	Y	Y	N/A
pushpin	Y	Y	Y	Y	Y	Y	Y	Y	P	Y	Y	Y	Y	Y	Y	Y	Y	Y	P	P	N/A
janus	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	N/A	N/A	N/A	Y	Y	N/A
KrakenD	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	P	N	N	N/A	N/A	N	Y	N	N/A
Lura	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N	Y	Y	N	N/A	N/A	N/A	N	Y	N	N/A
Ambassador	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N/A	N/A	N/A	N/A	Y	N/A	N/A
WSO2 API Microgateway	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N/A	N/A	N/A	N	Y	N	N/A
Goku	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N/A	P	N	N/A	N/A	N/A	N/A	Y	P	N/A
Express Gateway	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N/A	Y	N	N/A	N/A	N/A	N/A	Y	P	N/A
Easegress	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N/A	Y	Y	N/A	N/A	N/A	N/A	Y	N/A	N/A
spring-cloud-gateway	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N	Y	Y	Y	N/A	N/A	N/A	Y	Y	N/A	N/A

SSL column represent data gathered when the script was run with the SSL protocol enabled; (iii) *Min* column represent a minimal measured resource usage; (iv) *Max* column represent a maximal measured resource usage; (v) *Mean* column represent a mean of all measurements; (vi) *Median* column represent a median value of all measurements. Each row in tables represents an API gateway solution with the suffix -cpu or -mem, depending on whether the CPU or RAM usage was measured.

When *pushpin* was tested on *Device 2* with the SSL protocol enabled, it was noted that *pushpin* cannot handle a high number of requests in parallel. This may be possible due to the weak hardware on *Device 2*. Since we could not run all sub-scenarios on *Device 2*, partially gathered data cannot be compared with data gathered when another API gateway was tested on *Device 2*. Therefore, data gathered from *Device 2* when *pushpin* was tested with the SSL protocol enabled are omitted from Table IV. As discussed in the previous section, we were unable to run *pushpin* on the Windows OS. Therefore, there are no measurements presented in Table V for the *pushpin* API gateway.

Based on the measured performances presented in Tables III through V, the *Tyk* API gateway is identified as the best candidate for applying it in embedded systems. As the outcome of the evaluation phase, we conclude that *Tyk* fulfills all mandatory requirements. The only optional requirement we were not able to check related to *Tyk* is whether it is a standalone webserver or not.

VI. CONCLUSION

Leaning towards applying an API gateway to support microservice architectures in embedded systems, we presented a structural specification of requirements for such a case. We used these requirements to evaluate API gateway solutions that may be used in embedded system. The requirement specification may be used by other researchers as a basis to choose an appropriate API gateway for their needs and can be modified and extended with new requirements if needed.

In this paper, we evaluated and compared different API gateway solutions applied in a small-scale industrial setup and discussed the evaluation results. One of the conclusions of the evaluation process is that there are existing API gateways that may be used in embedded systems, even if they are not primarily made to be used in such systems.

During the overview of API gateway solutions, the biggest issue was the lack of information we expected to find in the solutions' documentation. The documentation was needed when we were choosing which solutions will be tested in the small-scale industrial setup. We selected API gateways that fulfil mandatory requirements based on their documentation. However, we did not find an API gateway that fulfills all the identified requirements. Among the API gateways we tested in the small-scale industrial setup, the *Tyk* API gateway fulfilled most of the specified requirements and provided the best performances. Accordingly, we chose *Tyk* as the best candidate to be applied in a microservice architecture for embedded systems.

As we tested only standalone open-source API gateway solutions, it is possible that a commercial solution that fulfills all the identified requirements may exist. Also, commercial solutions may provide better performances than the ones we tested. Therefore, as a

TABLE II. EVALUATION OF FLASH USAGE

API Gateway	Usage (MB)
Kong Gateway	**~274**
tyk	**~77**
pushpin	**~48**
apisix	**~380**

TABLE III. EVALUATION STATISTICS FOR DEVICE 1

API Gateway	High-end Linux-like OS device							
	No SSL				SSL			
	Min	Max	Mean	Median	Min	Max	Mean	Median
tyk-cpu	9.100	20.000	10.220	10.000	9.100	60.000	16.750	10.000
tyk-mem	1.100	1.400	1.270	1.300	1.100	1.500	1.310	1.300
pushpin-cpu	9.100	80.000	18.190	10.000	9.100	130.000	34.970	20.000
pushpin-mem	2.000	2.300	12.195	2.300	2.000	2.800	2.370	2.400

TABLE IV. EVALUATION STATISTICS FOR DEVICE 2

API Gateway	Low-end Linux-like OS device							
	No SSL				SSL			
	Min	Max	Mean	Median	Min	Max	Mean	Median
tyk-cpu	5.000	150.000	15.680	7.700	5.600	200.000	62.700	14.300
tyk-mem	7.100	9.100	8.000	7.900	6.900	9.700	8.100	7.900
pushpin-cpu	5.900	233.300	63.480	42.800	/	/	/	/
pushpin-mem	13.300	15.100	13.950	14.200	/	/	/	/

TABLE V. EVALUATION STATISTICS FOR DEVICE 3

API Gateway	High-end Windows OS device							
	No SSL				SSL			
	Min	Max	Mean	Median	Min	Max	Mean	Median
tyk-cpu	2.500	39.000	8.840	5.500	2.500	50.000	10.920	5.500
tyk-mem	0.002	0.003	0.003	0.003	0.001	0.003	0.003	0.003
pushpin-cpu	/	/	/	/	/	/	/	/
pushpin-mem	/	/	/	/	/	/	/	/

future work, we are planning to include commercial solutions in the evaluation process as well and present their performances in our industrial setup.

ACKNOWLEDGMENT

This paper is supported by KEBA AG Linz and by the Ministry of Education, Science and Technological Development under grant number 451-03-68/2022-14/200156 "Innovative scientific and artistic research from the FTS (activity) domain".

REFERENCES

[1] A. Homay, A. Zoitl, M. de Sousa, and M. Wollschlaeger, "A Survey: Microservices Architecture in Advanced Manufacturing Systems," in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, 2019, vol. 1, pp. 1165–1168. doi: 10.1109/INDIN41052.2019.8972079.

[2] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*, 1st ed. O'Reilly Media, Inc., 2016.

[3] C. Richardson, *Microservices Patterns: With examples in Java*, 1st ed. Shelter Island, New York: Manning, 2018.

[4] G. Buchgeher, R. Ramler, H. Stummer, and H. Kaufmann, "Adopting Microservices for Industrial Control Systems: A Five Step Migration Path," in *Proceedings of 2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Vasteras, Sweden, 2021, pp. 1–8. doi: 10.1109/ETFA45728.2021.9613622.

[5] S. Wang, C. Du, J. Chen, Y. Zhang, and M. Yang, "Microservice Architecture for Embedded Systems," in *Proceedings of 2021 IEEE 5th Information Technology,Networking,Electronic and Automation Control Conference (ITNEC)*, Xi'an, China, 2021, pp. 544–549. doi: 10.1109/ITNEC52019.2021.9587154.

[6] M. Mathijssen, M. Overeem, and S. Jansen, "Identification of Practices and Capabilities in API Management: A Systematic Literature Review," *arXiv:2006.10481*, 2020, Accessed: Jan. 08, 2022. [Online]. Available: http://arxiv.org/abs/2006.10481

[7] J. T. Zhao, S. Y. Jing, and L. Z. Jiang, "Management of API Gateway Based on Micro-service Architecture," *J. Phys.: Conf. Ser.*, vol. 1087, p. 032032, 2018, doi: 10.1088/1742-6596/1087/3/032032.

[8] F. Montesi and J. Weber, "Circuit Breakers, Discovery, and API Gateways in Microservices," *arXiv:1609.05830*, 2016, Accessed: Jan. 08, 2022. [Online]. Available: http://arxiv.org/abs/1609.05830

[9] S. Gadge and V. Kotwani, "Microservice Architecture: API Gateway Considerations," GlobalLogic, Inc, 2017, p. 13.

[10] R. Xu, W. Jin, and D. Kim, "Microservice Security Agent Based on API Gateway in Edge Computing," *Sensors*, vol. 19, no. 22, Art. no. 22, 2019, doi: 10.3390/s19224905.

[11] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A Design Science Research Methodology for Information Systems Research," *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, 2007, doi: 10.2753/MIS0742-1222240302.