Mixed-Paradigm Framework for Model-Based Systems Engineering

Philipp Helle, Stefan Richter, Gerrit Schramm and Andreas Zindel
Airbus Central R&T
Hamburg, Germany
email: {philipp.helle, gerrit.schramm, stefan.richter, andreas.zindel}@airbus.com

***Abstract*—In a time when competition and market in aviation industry drive the need to shorten development cycles especially in early phases, both automation of processes and integration of tools become important. While constraints, such as make or buy decisions or corporate Information Technology (IT) governance influence the overall tool infrastructure in different directions, microservices are a fast-rising trend in software architecting. But that does not mean that the more traditional monolithic software architecture is dead. A resulting mixed-paradigm software applications can also be seen as an opportunity to profit from the best of both worlds. To support a newly developed complex system development approach called Smart Component Modeling, a supporting application framework prototype is subject to development with the objective to reduce both time and resources required during product development cycles. This paper describes the software architecture styles and deployment approaches that were used in a research project at Airbus for building a prototype and discusses challenges and opportunities that were encountered.**

Keywords–model-based systems engineering; microservices; REST.

I. INTRODUCTION

The MicroService Architecture (*MSA*) is a style that has been increasingly gaining popularity in the last few years [1] and has been called "one of the fastest-rising trends in the development of enterprise applications and enterprise application landscapes" [2]. Many organizations, such as Amazon, Netflix, and the Guardian, utilize *MSA* to develop their applications [2].

Pursuing the notion that "Microservices aren't, and never will be, the right solution in all cases" [3], this paper describes the architecture and development approach that was used in a research project at Airbus for building a prototype application framework for Model-Based Systems Engineering (*MBSE*). According to The International Council on Systems Engineering (INCOSE), "Model-Based Systems Engineering (*MBSE*) is the formalized application of modeling to support system requirements, design, analysis, verification and validation, beginning in the conceptual design phase and continuing throughout development and later life cycle phases" [4]. This framework does not rely on a single paradigm but instead mixes different paradigms, viz. architecture patterns and deployment approaches, to achieve the overall goals: agility, flexibility and scalability during development and deployment of a complex enterprise application landscape.

This paper is structured as follows: Section II describes the modeling method that the built prototype *MBSE* framework is supposed to support. Section III provides background information regarding the different enterprise application architecture paradigms. Section IV explains the IT infrastructure in which the framework is deployed and Section V describes how and what features have been implemented in the prototype. Section VI discusses advantages and disadvantages of the mixed-paradigm approach. Section VII talks about the ongoing and future improvement effort before Section VIII wraps everything up with a conclusion.

II. SCM MODELLING METHOD

In [5], we provide a detailed account of the newly developed *MBSE* paradigm, called smart component modeling (*SCM*), that is rooted in a proposed change in the aircraft development process to include an out of cycle component development phase, in which components of an aircraft are developed independently of the traditional linear development process. These components are then adapted to the specific needs of a program within the more linear cycle. Furthermore, the paper describes a metamodel for modeling these so-called smart components based on proven *MBSE* principles [6]. Since the models are being defined outside of an aircraft program when requirements are not yet fixed, the models have to be parametric. An *SCM* is a self-contained model that can be developed out of cycle and enables capturing of all information relevant to the development of the component. *SCMs* are foreseen to be stored in a repository, called the *SCM Library*. This enables sharing and reuse. When the in-cycle phase of an aircraft or aircraft system development starts, the assets in the *SCM Library* are pulled and used as pre-defined and pre-verified components for a new development. The *SCM* metamodel defines all objects and their relations that are required to capture information related to smart components in models. The development of the *SCM* metamodel was driven by internal use cases and inspired by existing modeling languages such as the Systems Modeling Language (*SysML*) [7].

The requirements for the methodology supporting this new out of cycle process were as follows:

- The methodology shall be based on *MBSE* principles.
- The methodology shall be independent from any specific application domain.
- The methodology shall enable a product-line oriented product development, i.e., the metamodel must allow modeling of different variants of a product and ensure a consistent configuration and parametrization.
- The methodology shall enable inclusion of already existing domain models, i.e., models in a domain-specific modeling language.
- The methodology shall enable automatic verification of models, i.e., it shall be possible to check if the built models adhere to the modeling paradigm and to user-defined constraints.
- The methodology shall enable consistent modeling not only of the product itself but also of the context,

 ISBN: 978-1-61208-750-4

such as the industrial system used to built the product and allow the creation of relationships between the modeled artifacts.

The requirements for the application framework supporting this new modeling paradigm are as follows:

- The application framework shall be deployable in the current corporate IT infrastructure
- The application framework shall allow a heterogeneous technology stack to deliver the best solution for a designated purpose.
- The application framework shall be scalable with increasing number of models and users.
- The application framework shall be scalable in terms of model calculation performance.
- The application framework shall support continuous deployment strategies and agile frameworks to enable fast delivery and high flexibility.
- The application framework shall be efficient with regards to computing resources and reduce the companies ecological footprint.

III. Architecture paradigms

This section provides background information regarding the two main architecture paradigms that are used today: monolithic software and *MSA*. Service Oriented Architecture (*SOA*) and serverless architecture [8] are not described in detail as *SOA*, especially from a deployment perspective, still resembles monolith software [9] and serverless can be seen as taking *MSA* one step further [10].

A. Monolithic software

[11] defines a monolith as "a software application whose modules cannot be executed independently". This architecture is a traditional solution for building applications. A number of problems associated with monolithic applications can be identified:

- Due to their inherent complexity, they are hard to maintain and evolve. Inner dependencies make it hard to update parts of the application without disrupting other parts.
- The components are not independently executable and the application can only be deployed, started and stopped as a whole [12].
- They enforce a technology lock-in, as the same language and framework has to be used for the whole application.
- They prevent efficient scaling as popular and non-popular services of the application can only be scaled together [13].

Nevertheless, monolithic software is still widely used and, except for green-field new developments, there is hardly a way around it. [14] notes that a monolithic architecture is "often a more practical and faster way to start". Furthermore, if software from external parties is involved in a tool chain, it is not possible to change its architecture style.

B. Microservices

There is no single definition of what a *MSA* actually is. A commonly used definition by Lewis and Fowler says it is "an approach for developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API" [15]. Microservices typically consist of stateless, small, loosely coupled and isolated processes in a "share-as-little-as-possible architecture pattern" [16] where data is "decentralised and distributed between the constituent microservices" [17].

The term "microservices" was first introduced in 2011 [15] and publications on architecting microservices are rapidly increasing since 2015 [18]. In 2016, a systematic mapping study found that "no larger-scale empirical evaluations exist" [19] and concluded that *MSA* is still an immature concept.

The following main benefits can be attributed to *MSA*:

- Relatively small components are easier for a developer to understand and enable designing, developing, testing and releasing with great agility.
- Infrastructure automation allows to reduce the manual effort involved in building, deploying and operating microservices, thus enabling continuous delivery [18].
- It is less likely for an application to have a single point of failure because functionality is dispersed across multiple services [9].
- *MSA* does not require a long-term commitment to any single technology or stack.

[3] notes the obvious drawback of the current popularity of microservices that "they're more likely to be used in situations in which the costs far outweigh the benefits" even when monolithic architecture would be more appropriate.

In a study regarding the challenges of adopting microservices, [2] lists the distributed nature of *MSA*, which leads to debugging problems, the unavailability of skilled developers with intimate knowledge of *MSA* and finding an appropriate separation into services.

IV. Deployment Infrastructure

Corporate information technology (IT) environments imply very strict regularities when it comes to hard- and software architectures and deployments. Bringing in innovation in such an environment requires following a heterogeneous approach.

While it is more challenging to adapt hardware in a corporate context to cope with the latest innovations, service and software developments, e.g., ARM (Advanced RISC Machine) CPU (Central Processing Unit) platform based servers, GPU (Graphics Processing Unit) assisted computing or wide-usage of FPGAs (Field Programmable Gate Arrays), the application platform layer adaption is typically less demanding because almost any state-of-the-art deployment form, like bare-metal, Infrastructure-as-a-Service (*IaaS*), Platform-as-a-Service (*PaaS*) or Function-as-a-Service (*FaaS*) can be rolled out on standard server hardware.

The rationale for choosing a specific deployment form is based on various constraints imposed by corporate policies and long-term strategy decisions:

- Is the envisaged deployment form available in the corporate infrastructure?

 ISBN: 978-1-61208-750-4

- Has the deployment form limitations due to corporate policies, e.g., restricted internet access, restricted repository access?
- Are there any license limitations?
- Are there geolocation limitations for certain services, e.g., in a multinational company with multinational regulations according to law?
- Is the service available on premise or only on public cloud?
- Does a deployment form for a particular service fit in the long-term corporate IT strategy, e.g., make or buy decisions?

For the *Smart Component Modeling* prototype, it was necessary to make use of a heterogeneous software and hardware infrastructure provided by the corporate IT. Therefore, the deployment took place on *IaaS*, *PaaS* and *FaaS* platforms. Also, end user devices are involved, for example for running the *SCM workbench* (see Figure 2). That variety of platform types was chosen to provide inside information on how a new engineering concept could be supported by different software architecture approaches to be efficient in terms of development time, continuous integration (CI), resource efficiency and scalability.

A. Infrastructure-as-a-Service

In the context described above, *IaaS* is used to describe a hosting platform based on bare-metal and hosted hypervisors. It provides a variety of virtualized operating systems that are in compliance with corporate IT regulations.

For the prototype, the services hosted on classical virtual machines are mainly databases used as persistent layers for distributed Web applications. The main reason for not hosting the Web applications together with their respective persistence layer are resource restrictions. Current company policies prevent external access to the databases if they are part of the same microservice image as the hosting environment. This would either limit database management to a Web-based command line interface or require the implementation of a Web service deployed in the same container. Also, other external services could not be used to access the databases. This limitation is purely based on a decision made by the company's IT governance, but reflects day to day reality in corporate environments.

For any other Web application around the *SCM* prototype development, *IaaS* was avoided as the resource overhead cannot compete with *PaaS* or *FaaS*.

B. Platform-as-a-Service

In the following section, *PaaS* refers to an on-premise deployment of the *Red Hat Openshift*[20] platform. It is a platform built around *Docker*[21] containers orchestrated and managed by *Kubernetes* on a foundation of *Red Hat Enterprise Linux*.

In the prototype, *PaaS* plays a critical role for the continuous integration strategy. The image format used for the deployments follows the Source-to-image (*S2I*) concept. *S2I* is a toolkit and workflow for building reproducible container images from source code [22]. *S2I* produces ready-to-run images by injecting source code into a container image and letting the container prepare that source code for execution. The source code itself is hosted on an on-premise Github Enterprise[23] instance and the dependent resources are provided via an on-premise *Artifactory*[24] deployment that reflects the official sources of the required development environment such as Maven[25], npm, Python or NuGet.

The whole continuous deployment chain is secured via an exchange of keys and certificates to prevent disruptions for example due to company introduced password cycles for the developer and deployment accounts. The deployment speed is improved by using system instances for the *S2I* chain in the same geolocation of the company to prevent larger inter-site data transfers and round-trip times.

The microservice concept, together with *PaaS*, allows a massive reduction of resource allocations compared to an *IaaS* deployment, especially if the services are single and independent Web applications.

There are still limitations in the corporate environment that currently prevent larger scale use of the technology. The current setup allows a limited number pods per node, which becomes an issue when a service uses the scaling capability of the *OpenShift* platform. A second limitation is linked to the allocated sub-network and the deployment of the platform. All inter-service communication is routed via a unique company internal network. The *PaaS* instance does not re-use a network range that is already present in the company for inter-service communications as it would impose other challenges regarding communication from within the *PaaS* instance towards other company services. The rationale for the chosen *PaaS* implementation is primarily the reduction of classical virtual machines for simple hosting jobs and only secondary the creation of a massively scalable infrastructure for new service applications.

To cope with these limitations the prototype furthermore reduces the deployment footprint of single services for certain applications as described below.

C. Function-as-a-Service

FaaS is used for tiny stateless jobs, e.g., rendering of images. These services are monitored by an orchestrator that decommissions containers after idling for a defined time. This reduces resource usage further and has advantages in a scenario with a larger number of services.

The deployment architecture of the *FaaS* instance allows launching service containers within milliseconds. The applied software stack is *OpenFaaS* based on *Docker Swarm* running on a *Debian*[26] virtual machine.

One *FaaS* instance consumes resources similar to a pod on the above mentioned *PaaS* environment and hosts numerous services without performance limitations. While *PaaS* exposes containers under their distinct IP (Internet Protocol) addresses, *FaaS* comes with a reverse proxy that hides all containers and requires less IP addresses. This reduces the effort for routing name resolution and their documentation.

V. Implementation and Integration

The implementation of the prototype framework is split into different logical bricks as depicted by Figure 1. The *Architect Cockpit* allows a system architect to use existing models, to schedule the execution of simulations and to review results.

 ISBN: 978-1-61208-750-4

The *SCM Workbench* enables SCM developers to create and version *SCMs*. The *Back End* provides different services such as the orchestration of different processors to perform the execution of simulations.

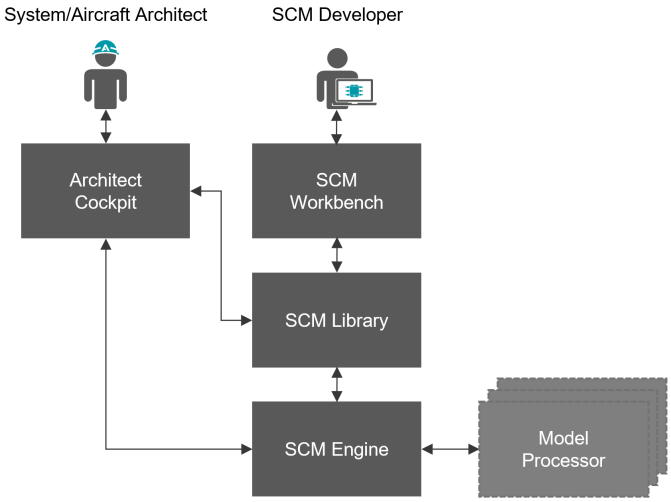

Figure 1. Prototype tool overview

A. Architect Cockpit

In order to reduce the workload and make the work for the architects as convenient as possible the interface for the cockpit is setup as an Angular single-site application. This allows using this entity without installing custom software and without bothering the user with update and migration procedures. The site is built using a Jenkins pipeline and then deployed on a specific git repository branch. A webhook on this branch triggers an *OpenShift* instance to build an Express.js server serving the previously build site on a *PaaS* cluster.

From a functional point of view the *Architect Cockpit* gives a reduced view on *SCMs*. Only information, which is necessary for the work of an architect is available and can be modified. This results in a nearly full intuitive usage of the interface and prevents faulty configurations. For example, some parameters can only be changed within a certain range. Ranges are defined by the model developer who knows the limitations best. The architect does not need to have a deep understanding of these limitations when using the predefined models.

B. SCM Workbench

The *SCM Workbench* is a full-fledged graphical editor to work with *SCMs* implemented as a monolithic rich-client application. It is implemented in an Eclipse Rich Client Platform *(RCP)* and based on the Eclipse Modeling Framework *(EMF)* [27]. It is a modeling framework and code generation facility for building tools and other applications based on a structured data model. *EMF* provides tools and run-time support to produce a set of Java classes from a model specification, along with a set of adapter classes that enable viewing and editing of the model, and a basic editor.

EMF is the basis for the Obeo Designer tool[28], which builds on the Eclipse Sirius project [29] and allows definition of graphic editors based on a defined *EMF* metamodel. This enables rapid prototyping of modeling solutions, which is ideal for a research/prototyping environment such as Airbus Central R&T. Changes to the metamodel are almost instantly available in the *SCM Workbench*, our prototype *SCM* modeling tool. On the other hand, *EMF* and *Obeo Designer* are mature and have been proven in industrial practice, e.g., *Capella*, the modeling tool from Thales that implements the *Arcadia* method is built with *EMF* and *Obeo Designer* as well [30].

Using such a rapid prototyping approach for the *SCM Workbench* can be easily misunderstood as just a proof-of-concept study. The final look and feel of the graphical editor for the *SCMs* is only limited by the amount of development time used for UX polishing. The workflow and information accessibility as well as the connection to a versioning system is comparable to other commercially available modeling tools, which are well known by the developers. It is assumed that a SCM developer has to take a short on-boarding training before using the *SCM Workbench*.

C. Back End

The *Back End* is build from several different entities that are based on different paradigms. These entities are described in the following paragraphs.

1) SCM Library: The *SCM Library* stores the models that have been created using the *SCM Workbench*. It is based on Connected Data Objects *(CDO)* a Java model repository for *EMF* models and metamodels. The specific implementation in use is the *Obeo Designer Team Server (ODTS)* which enables concurrent engineering of *EMF* models. A custom plug-in allows other services and applications to access the model repository through a *REST* interface. Due to its complex deployment strategy the *SCM Library* is deployed in an *IaaS* environment which allows more user interaction during updates.

2) SCM Engine: The *SCM Engine* can interpret *SCMs*, check constraints and run parametric calculations either as a single simulation run or as a *Design of Experiments* setup with multiple samples. It is a Java application executed in an OpenJDK Virtual Machine. Access to the engine is established through *REST* interfaces that are hosted on a Jetty server. The endpoints are described and documented using the Jersey framework. The *SCM Engine* is hosted on a *PaaS* instance and allows rolling updates, automated builds and scaling.

3) Model Processors: The *Performance Model API* serves as a glue between external domain-specific models with their own solver or simulation engine and the *SCM Engine*. A *Model Processor* is an application that implements this API to execute a specific model type. The API enables the *SCM Engine* to orchestrate simulations tools in a unified way and guides developers through the process of integrating additional simulation tools into this environment. In order to include a new model type in the *SCM* application framework, a model type specific *Model Processor* has to be implemented that implements the *Performance Model API* and connects to the model type specific solver or simulator. A reference implementation shows how this works for Excel models. An Excel model is processed by a Java application running in an OpenJDK VM using the Apache POI framework. Depending on the type of model and, e.g., the license and installation requirements of the model solver or simulator, the *Model Processor* can be deployed in any of the available deployment options *IaaS*, *PaaS* and *FaaS*.

 ISBN: 978-1-61208-750-4

Figure 2 depicts how the components of the *SCM* tool framework prototype are deployed in our infrastructure.

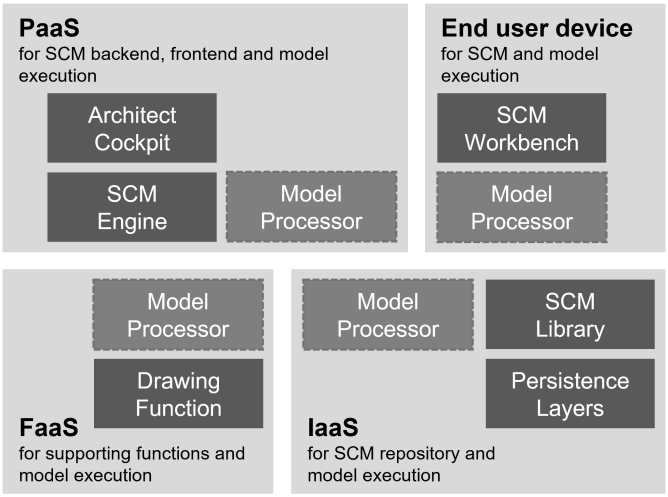

Figure 2. Prototype tool deployment

To make the polyglot approach of the *MSA* work and integrate each service all participating entities need to agree to a commonly understood interface. For the prototype *Representational State Transfer (REST)* over HTTP was chosen as the default interface combined with JavaScript Object Notation (*JSON*) as serialization format. *REST* over HTTP is a de facto standard since almost every technology stack provides at least an HTTP API if not specialized *REST* frameworks and clients such as *JAX-RS*. *JSON* as a serialization format is accepted and provides solid tooling on all integrated technologies. In addition many front-end frameworks natively support *JSON* such as *JavaScript* or *Ruby*. This eases the integration work needed to be done for the implementation of our demonstrators mainly the *Architect Cockpit*. As an added bonus it is easily digestible by human user, which helped tremendously with debugging. To built up process chains utilizing the deployed microservices we selected *Node-RED*. It provides all the tools necessary to handle *HTTP* based *REST* APIs and *JSON* based message bodies and is integrated well into the existing environment.

VI. Evaluation

Evaluating the mixed-paradigm approach, we experienced that developers where able create a working deployment much faster compared to the traditional approach using virtual machines. This also includes the amount of times that a new version of the service was built from once a week to several times a day using the automated CI pipeline. This increased the general development velocity as well as the prototypes feature set, which helped us to tailor the application to our stakeholder needs.

The raised deployment speed increased the number of times we experienced broken client applications. This was due to a violated interface contract between the services if the new features where not integrated properly. A well defined and adhered to interface specification is paramount for the success of introducing this mixed-paradigm approach.

In general, we noticed a greater sense of ownership of single developers over their service/code, which lead to a hike in the overall implementation quality. The mandatory usage of the git version control system increased the maintainability of the code base. The combination of git and the *Openshift* framework made it easy to recover from failures and faulty builds, which lead to a constant up-time of all services. In the future the introduction of additional agile software development principles like *Test Driven Development* could further increase the code quality.

The mixed-paradigm approach that was used to develop and deploy the prototype discussed in this paper led to reduced complexity, lower coupling, higher cohesion and a simplified integration. This in turn enabled agile collaboration for continuous delivery and integration of the solution.

VII. Outlook and future challenges

In the previous sections, we described how *MSA* can support the chosen polyglot approach utilizing a variety of different technology stacks and storage solutions. This enabled us to select the most fitting technical solution for the required functionality. Additionally the network based architecture provides an environment that is well suited for a multinational company like Airbus with sites scattered throughout different sites and IT domains. It also provided a commonly understood deployment layer for our cross-functional project team.

MSA supports us with the agility and velocity needed to convince our customers of our approach and implement a prototype that can handle the complexity of our *SCM* modeling approach. However, during the development we found stumbling blocks that need awareness once the scale changes from a research project prototype to a full scale industrial roll out.

Corporate IT – The proposed environment builds and hosts microservices in an agile and automated way. This requires the setup and maintenance of a CI pipeline (in our case *Openshift*/*GitHub*), which results in additional costs as well as an IT department that is capable of dealing with those investments. Additionally setting up certificate chains and firewalls to allow for secure communication inside the corporate network need to be accounted for. On the developer side roadblocks like proxy server hindering communication and enabling cross-origin resource sharing (*CORS*), which allows for communication between different domains need to be taken care of.

Service discovery – Once we reached a critical mass of microservices environment we discovered that it is hard to keep track of what services have already been implemented and what functionality each service provides. Even in our research project this point was reached rather quickly. Thus we introduced *Swagger*[31] as a Web based documentation for all our services and implemented a simple dashboard where services could be registered against. This allowed for manual service discovery across the team. In the future automated service discovery through bots and processable service descriptions will bring more value to the *MSA* approach by handling the sprawling service environment.

Now that we optimized the CI pipeline in the first half of the project we experience a rapid increase in deployed services. This allowed us to swiftly introduce new functionality as microservices, boosting the capabilities of our proof of concept prototype. It shows that *MSA* can initially speed up the

 ISBN: 978-1-61208-750-4

implementation velocity of a new project. Once we continue with the project more efforts will go towards managing the volume of services as well as (network) performance and reliability.

VIII. Conclusion

A direct, specific and measurable comparision between the described mixed-paradigm and a classical approach is not possible as it would have required the same infrastructure landscape to have been developed and deployed multiple times using different concepts. Nevertheless implementors were given the freedom to decide for every distinct artifact to freely choose the paradigm used for implementation. Furthermore developers were allowed to spilt artifacts which enables to select the right paradigm for each problem within. Later the interface documentation allowed the developers to easily re-implement an artifact using a different paradigm in case the initial decision for a specific paradigm reveals to have been not an optimal choice. Therefore the selection of the right paradigm appears to be inherent and native. To support a newly developed *MBSE* approach called Smart Component Modeling, a supporting application framework prototype had to be developed. Instead of a single architecture and deployment paradigm, a mixed-paradigm approach was followed to take the advantages of the different options and to consider external constraints coming from the IT governance. The software bricks were implemented in monolithic, *SOA*, microservice and serverless architecture glued together by *REST* interfaces over HTTP. The deployment took place on Desktop-PC, *IaaS*, *PaaS* and *FaaS* platforms. It provided insight into how a new engineering concept could be supported by different software architecture approaches to be efficient in terms of development time, continuous integration, resource efficiency and scalability.

References

[1] N. Dragoni et al., "Microservices: yesterday, today, and tomorrow," in Present and ulterior software engineering. Springer, 2017, pp. 195–216.

[2] J. Ghofrani and D. Lübke, "Challenges of microservices architecture: A survey on the state of the practice." in ZEUS, 2018, pp. 1–8.

[3] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," IEEE Software, vol. 35, no. 3, 2018, pp. 24–35.

[4] D. D. Walden, G. J. Roedler, K. Forsberg, R. D. Hamelin, and T. M. Shortell, Eds., Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities, 4th ed. Hoboken, NJ: Wiley, 2015.

[5] P. Helle, S. Feo-Arenis, A. Mitschke, and G. Schramm, "Smart component modeling for complex system development," in Proceedings of the 10th Complex Systems Design & Management (CSD&M) conference, forthcoming.

[6] A. Reichwein and C. Paredis, "Overview of architecture frameworks and modeling languages for model-based systems engineering," in Proc. ASME, 2011, pp. 1–9.

[7] Object Management Group, OMG Systems Modeling Language (OMG SysML), v1.2. OMG, Needham, MA, 2008.

[8] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "Serverless programming (function as a service)," in 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). IEEE, 2017, pp. 2658–2659.

[9] A. Karmel, R. Chandramouli, and M. Iorga, "Nist definition of microservices, application containers and system virtual machines," National Institute of Standards and Technology, Tech. Rep., 2016.

[10] I. Baldini et al., "Serverless computing: Current trends and open problems," in Research Advances in Cloud Computing. Springer, 2017, pp. 1–20.

[11] N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, "Microservices: Migration of a mission critical system," arXiv preprint arXiv:1704.04173, 2017.

[12] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, "From monolithic to microservices: an experience report from the banking domain," Ieee Software, vol. 35, no. 3, 2018, pp. 50–55.

[13] M. Villamizar et al., "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in 2015 10th Computing Colombian Conference (10CCC). IEEE, 2015, pp. 583–590.

[14] ——, "Cost comparison of running web applications in the cloud using monolithic, microservice, and aws lambda architectures," Service Oriented Computing and Applications, vol. 11, no. 2, 2017, pp. 233–247.

[15] M. Fowler and J. Lewis. Microservices a definition of this new architectural term. [Online] http://martinfowler.com/articles/microservices.html [Accessed: 11 September 2019].

[16] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: current and future directions," ACM SIGAPP Applied Computing Review, vol. 17, no. 4, 2018, pp. 29–45.

[17] D. Shadija, M. Rezai, and R. Hill, "Towards an understanding of microservices," in 2017 23rd International Conference on Automation and Computing (ICAC). IEEE, 2017, pp. 1–6.

[18] P. Di Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: trends, focus, and potential for industrial adoption," in 2017 IEEE International Conference on Software Architecture (ICSA). IEEE, 2017, pp. 21–30.

[19] C. Pahl and P. Jamshidi, "Microservices: A systematic mapping study." in CLOSER (1), 2016, pp. 137–146.

[20] RedHat, "Openshift," https://www.openshift.com/, 2019, [Online; accessed 21-October-2019].

[21] Docker Inc., "Docker," https://www.docker.com/, 2019, [Online; accessed 21-October-2019].

[22] A. Lossent, A. R. Peon, and A. Wagner, "PaaS for web applications with OpenShift origin," Journal of Physics: Conference Series, vol. 898, oct 2017, p. 082037.

[23] GitHub, Inc., "Github," https://github.com/, 2019, [Online; accessed 21-October-2019].

[24] JFrog Ltd, "Artifactory," https://jfrog.com/artifactory/, 2019, [Online; accessed 21-October-2019].

[25] The Apache Software Foundation, "Maven," https://maven.apache.org/, 2019, [Online; accessed 21-October-2019].

[26] Software in the Public Interest, Inc., "Debian," https://www.debian.org, 2019, [Online; accessed 21-October-2019].

[27] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, EMF: eclipse modeling framework. Pearson Education, 2008.

[28] Obeo, "Obeo designer," https://www.obeo.fr/en/, 2019, [Online; accessed 21-October-2019].

[29] V. Viyović, M. Maksimović, and B. Perisić, "Sirius: A rapid development of dsm graphical editor," in IEEE 18th International Conference on Intelligent Engineering Systems INES 2014. IEEE, 2014, pp. 233–238.

[30] P. Roques, "MBSE with the ARCADIA Method and the Capella Tool," in 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016), Toulouse, France, Jan. 2016.

[31] SmartBear Software, "Swagger," https://swagger.io/, 2019, [Online; accessed 21-October-2019].

 ISBN: 978-1-61208-750-4