Implementing and Deploying an Execution Environment for Multidisciplinary-Analyses in a Heterogeneous Tool Landscape

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

Abstract—**More complex products, shorter product lifecycles, as well as a faster entry into the market forces engineering departments to develop and deploy new strategies. Model-based Systems Engineering (MBSE) is thought to play a vital role to master the current and future challenges. MBSE provides its full benefits when the models are not only used descriptively but also analytically. The execution of multidisciplinary analyses in a heterogeneous tool landscape requires the development and deployment of an analysis execution environment. Based on microservices, message bus technology and a centrally managed data exchange format it was possible to build a set of applications that are efficiently extendable, maintainable and scalable. The paper describes the implemented execution environment and provides feedback on the technologies that were used.**

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

I. Introduction

As systems and products become more integrated and more complex engineering departments seek strategies to cope with their complexity. MBSE is recently gaining popularity for the engineering of complex systems. In the majority of applications, MBSE is used descriptively rather than analytically. While descriptive modelling is an important step towards a formal representation of the information about a system [1], re-using the same models for analysis provides additional benefits. These benefits include a reduction of repetitive work due to re-use and avoidance of error-prone human interaction while model processing by using a formalized machine-readable format.

To illustrate the need for evaluating different engineering solutions we take the example of a regional aircraft. This aircraft shall be analyzed regarding its propulsion system. Two options are foreseen, either electric propulsion or jet fuel propulsion. Both solutions require a wide-range of physical modelling. Different engineering disciplines typically use different tools for their analysis. The complexity of the system components and disciplines, in this case for an aircraft, cannot be properly addressed by a single tool (e.g Simulink/Matlab, Dymola) or even a tool suite (e.g. ModelCenter, 3DEXPERIENCE). Some disciplines use optimized solvers encapsulated in business owned tools, that do not naturally integrate with Commercial off-the-shelf (COTS) solutions. Enabling the execution of multidisciplinary analyses therefore requires the development and deployment of an execution environment that integrates this heterogeneous set of tools.

This paper provides a description of a software architecture for executing multidisciplinary-analyses that are described in Systems Modeling Language (SysML) models in a heterogeneous tool landscape. It is, in spirit, a continuation of the work described in [2]. The major differences are:

- The use of message queuing and a publish/subscribe pattern for service integration
- Using centrally defined message schemata to ensure interface consistency
- Extensive use of containers and templates to improved horizontal scaling
- A new metamodel based on the SysML standard instead of a proprietary Domain-Specific Language (DSL) to ensure integration into the corporate tool landscape

This paper is structured as follows: Following this introduction, Section II provides background information regarding descriptive MBSE and the concept of Parametric Analysis Definition Model (PAM). In Section III, the paper explains the implementation of a computation chain for PAMs and its components. Next, in Section IV it discusses and evaluates the advantages of the described implementation. This is followed by a brief outlook on challenges of automation and eco-efficiency ahead in Section V. Section VI concludes the paper and summarizes the most promising changes that have been made compared to previous work.

II. Motivation

The SysML tools nowadays typically come with the capability for model execution based on Activity Diagrams and statecharts as well as the possibility of evaluating Parametric Diagrams. These capabilities have been proven helpful to verify the models and validate their behavior. However, they are typically executed in the modelling tool running on a desktop machine. This often leads to challenges regarding scalability and reproducibility and the desktop computer is typically blocked for the duration of the analysis, which renders long-running tasks with a runtime of several days unfeasible. Additionally, the integration of external tools requires writing and adding code into the model, which requires programming skills that modelling experts do not necessarily have.

SysML's strength is to describe and capture many aspects of complex systems, their parts and their interdependencies. When it comes to modelling, simulating and optimizing the details in various disciplines, specialized domain-specific tools do this job better and are already well-established in the industry. The analysis of the regional aircraft with the two propulsion options requires not only specific COTS tools to perform the calculations for the electrical system or the jet propellant system, but also in-house tools for costs or weight distribution. In order to enable the selection of the best tool for each discipline, domain-specific tools have to be integrated into a computational chain, where data is sent from one tool to another to provide an overall analysis. To benefit from the strength of modelling, e.g., to make hidden/implicit knowledge explicit and reviewable, this kind of computational chain or analysis, is best captured in a model as well.

To achieve this, [3] proposes a shift from purely descriptive MBSE models towards models that also contain a description of the analysis in the form of PAMs. A PAM is an extension of the SysML based on SysML Activity Diagrams that allows the description of a flow-based analysis process. A software architecture is required to actually interpret and execute these PAMs. Figure 1 shows a high level logical view of the architecture that is required.

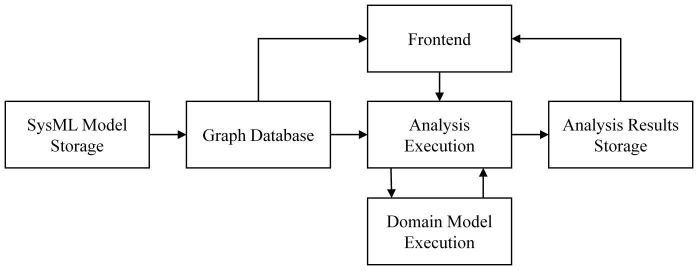

Figure 1. Architecture overview

SysML models are stored in a *SysML Model Storage* component that provides typical versioning functionality. However, access to this *SysML Model Storage* is slow and only possible using a proprietary Application Programming Interface (API). To alleviate this and to enable integrating information from other sources, a *Graph Database* component is used to store and integrate all relevant data. This database has both faster access mechanisms and allows more sophisticated querying using SPARQL Protocol and RDF Query Language (SPARQL). A web-based *Frontend* is provided for interaction with the user. It lists all PAMs stored in the database and allows the user to configure and start analyses. Once a PAM is selected and parameters are set the user can request the launch of an analysis through the *Frontend*. The *Analysis Execution* component is able to execute and analysis request by interpreting the PAM and following the flow-based analysis process. As part of the analysis execution, the *Analysis Execution* component has the capability to figure out, if and which external domain models need to be executed. It can trigger the execution of these domain models by specialized *Domain Model Execution* components and integrate the results back into the overall PAM execution. Once every part of an analysis has been executed, the analysis result is stored in an *Analysis Result Storage* component. The process is fully asynchronous, and the user receives a notification once a new result is available to be displayed through the *Frontend*.

III. Implementation and Integration

In the previous section the logical architecture of the PAM analysis execution environment was described. This section further explains how these logical components where implemented, integrated and deployed. To address the large scope challenges outlined in Section II it is obvious that such an application cannot be efficiently implemented as a monolithic software. The Micro Service Architecture (MSA) is a style that has been increasingly gaining popularity in the last few years [4] and has been called 'one of the fastest-rising trends in the development of enterprise applications and enterprise application landscapes' [5]. Many organizations, such as Amazon, Netflix, and the Guardian, utilize MSA to develop their applications [5]. Adopting the MSA paradigm, leads to an infrastructure with many services that may evolve over time. Figure 2 depicts the services that have been implemented.

The different services perform different tasks. Depending on the type and the context of each task different programming languages are suitable for implementing the service that has to perform the task. The MSA paradigm allows to choose the programming language for implementing a given service that is best suited for the task.

A typical sequence of tasks for a PAM analysis consists of the following steps:

- The user requests an analysis through the *User Interface*.
- The *User Interface* emits an analysis request message on Pulsar through the *Reverse Proxy* and the *Pulsar Gateway*.
- The *Experiment Executor* receives the message and loads the model that is the basis of the analysis request through the *Marklogic Connector*.
- The *Experiment Executor* interprets the model and determines which domain models are involved in the trade study.
- The *Experiment Executor* emits execution requests for each involved domain model.
- The *Task Router* receives each domain model execution request and forwards it to a *Domain Model Processor* that is suited to perform the request.
- The *Domain Model Processors* perform their requested computations and emit a result message.
- The *Experiment Executor* collects all results, computes the overall analysis result and emits one message for storing the result and one message for notifying the *User Interface* that a new result is available.
- The user receives the notification and is able to view the analysis result that was loaded through the *Cassandra Connector* in the *User Interface*.

Processing more complex models for an analysis and supporting additional user needs can easily lead to an increased number of services, e.g., to solve different domain specific models or provide new result analysis features. To cope with such scenarios it is important to implement measures to increase efficiency of maintainability, extensibility, upgradability and scalability of the overall architecture. Furthermore, the strategy used by the development team to implement the software services needs to be efficient, so that new services can be developed and integrated quickly. The most notable enablers with the highest impact on a successful and flexible software implementation are:

- End-User Interface
- Inter-service messaging

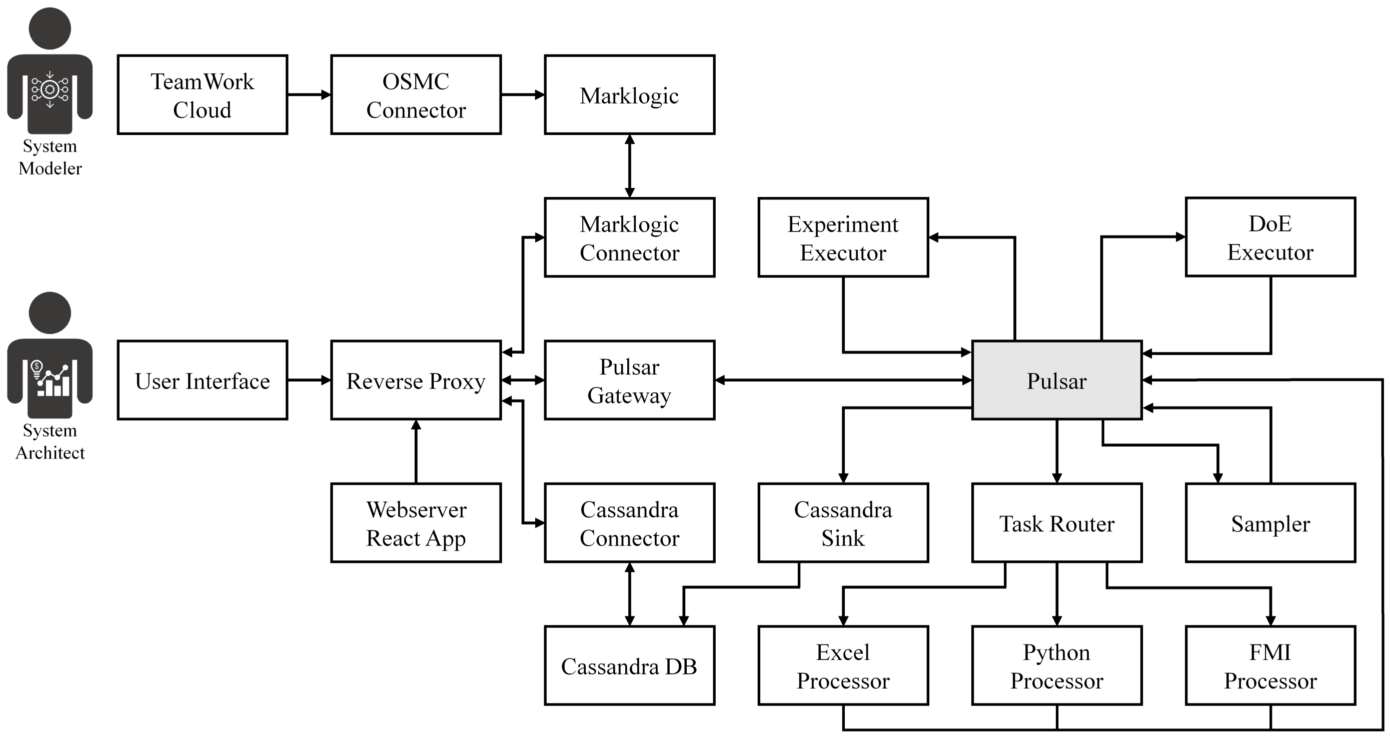

Figure 2. Detailed Architecture

- Interfacing concept
- Service Deployment

The following subsections focus on the key characteristics, which are influencing the technology concept selection.

A. User Interface

The *User Interface* is an important brick of the whole application as it needs to hide the functional complexity of the backend services and infrastructure from the end-user. It needs to be as flexible as the backend to adapt to new or changed functional requirements. As Figure 2 shows, the *User Interface* on the left side is the sole interface to the end-user, i.e., the *System Architect*. It is made up from the following different components.

- A *Reverse Proxy* to fulfill corporate security requirements.
- A *Webserver* hosting a React application to fulfill corporate design principles.
- Two database connector services, the *Marklogic Connector* and the *Cassandra Connector* that provide access to persistent data.
- The *Pulsar Gateway* that provides direct access to the broker for interaction with the analysis backend

B. Brokering and Message Queuing

When the user dispatches an analysis request several tasks are triggered on different services. In order to coordinate this, the architecture requires *Brokering and Message Queuing*.

Brokering is required to allow services to dynamically connect to a service bus without the emitting service knowing the addresses of each connected service a priori. This reduces the effort for configuration management of the service landscape drastically and enables horizontal scaling. *Message Queuing* ensures in a scaled environment that no task is being missed or computed twice. All services publish their requests on the message bus The message bus routes the messages to the services that have a subscription for it. In case a service is currently not availability to receive a message, e.g., due to a high workload or any temporal error, the message bus can queue and temporarily buffer the data until the service is accepting messages again. Depending on the selected Quality of Service (QoS) level, messages can be sent to several recipients in parallel or just to one for example.

Apache Pulsar, which offers both *Brokering* and *Message Queuing* has been chosen as the implementation for this for a number of reasons:

- Quality of Service (strong ordering and consistency guarantees)
- Horizontal scalability (millions of independent topics and millions of messages published per second)
- Load balancing
- Pulsar functions for implementing lightweight compute processes
- Pulsar IO for connecting to other external services such as databases
- Client API with bindings for different programming languages

C. Service Implementation Examples

To show that different reasons influence the choice of programming language for implementing the services, this subsection explains the rationale for choosing a programming language for a subset of services.

The *Experiment Executor* service is implemented in Java as some of the libraries that are used for handling the PAM interpretation such as the graph-theory based cycle detection exist in Java. The *Sampler* service that provides statistical methods for generating random data samples is written in Python because it is based on OpenTurns, which is exclusively available in python. The *Marklogic Connector* and the *Cassandra Connector* are programmed in TypeScript as they provide data access in JavaScript Object Notation (JSON) format through Hypertext Transfer Protocol (HTTP). TypeScript is based on JavaScript and has native support of JSON, well maintained packages for connecting to Cassandra and Marklogic databases and integrated asynchronous web server modules available.

An efficient service landscape consists of services written in different languages to enable exploiting the strengths of each language. This results in interfaces between services implemented in different languages and requires a mechanism to keep these interfaces coherent. Protocol Buffers [6] was chosen for this.

D. Protocol Buffers

Providing a language independent definition of each exchanged message that can be translated into different languages reduces the effort required for implementing the services and helps debugging. It is important for the message definition to provide serialization and deserialization functionality as well as compatibility with all different programming languages that are used. Googles Protocol Buffers promise to accomplish that. In comparison to JSON or Extensible Markup Language (XML), Protocol Buffers have a binary encoding resulting in a smaller footprint on the network. This is important as large analysis with many domain model invocations require many large messages to be transmitted.

E. Containers

As already explained, different languages are required to build the service landscape shown in Figure 2. Each service requires a different runtime environment and is required to scale horizontally individually, i.e., some services are required more often than others. There are several options to automatically deploy and maintain these services. Type 1 hypervisors like Kernel-based Virtual Machines (KVM), Xen or Elastic Sky X integrated (ESXi) in conjunction with automation tools like Ansible or Terraform are common solutions to roll out and maintain complex software architectures. Virtual machines running on hypervisors can be used to run the services, but the resource costs are high, and it requires that service developers have a deep knowledge of the underlying operating system to deploy their services.

A comparably lightweight alternative for service deployments are containers. The most common implementations are Linux Containers (LXC) or Docker. Both provide a promising set of features to support an end-to-end service automation scenario. Docker was chosen as service deployment platform for its wider community support, publicly available resources and features. The following list of Docker features were essential for choosing it as a deployment platform:

- Docker services are built from a series of layers. In Docker, each service consists of a base layer and every consecutive layer adds a piece of functionality. This allows Docker to reuse services with the same functionality across multiple services with different base layers [7].
- Docker uses a client-server architecture. This client-server architecture with the possibility to connect any public or private image registry offers new possibilities of distributed docker instances [8].
- The Docker swarm mode is Docker's native support for orchestrating clusters of Docker engines [9].
- Docker compose decreases the deployment effort for complex applications. It allows configuring service stacks consisting of multiple containers [10].
- Docker registry is a centralized store that keeps and provides Docker layers. An individual Docker Registry can be deployed locally or publicly available Docker registries like Docker Hub, Quay.io and Google Container Registry can be used [11].

The combination of these key characteristics offer a valuable support for a Continuous Integration (CI)/Continuous Deployment (CD) system.

IV. Evaluation

This section evaluates the design decisions made for implementing and deploying the software architecture. It explains how the key characteristics of the selected technologies made a difference with respect to scalability, message processing, maintainability and implementation efficiency. Scalability depends on several factors. One aspect is the better utilization of available computational resources by horizontally scaling services, and spreading functionality across multiple services. The later behavior has a strong link to implementation efficiency and maintainability, because smaller services with fewer lines of code require individually less maintenance effort. Additionally to its smaller profile of encoding/decoding and transmission message processing is also evaluated by the development time required to implement the message processing into a service using certain message processing paradigms.

A. Message Bus

The message bus connects all services relevant for performing PAM analyses. It is therefore a central element of the overall application as shown in Figure 2.

The *Experiment Executor* acts as an orchestrator for an analysis. It interprets the PAM, and executes it given the input variable values provided by the end-user. During this execution, it emits messages to trigger *Domain Model Processors*. When the *Domain Model Processors* finish their calculations the results are sent back and the *Experiment Executor* acts now as a subscriber. The signal on the message bus that a domain calculation result is available triggers a task to store it. A finished analysis task signals through the message bus to the *Frontend* that new results can be viewed or downloaded.

The architecture diagram shows only three different domain model processor services, i.e. Excel, Python and Functional Mock-up Interface (FMI). This is reduced list to exemplify the principle of an extendable pool of calculation services. In reality, more processors have been implemented and deployed.

Given this scenario, not using a message bus means that every new processor attached to the system results in a new peer-to-peer interface with the *Experiment Executor*. Furthermore, additional new interfaces would be required to signal the calculation status.

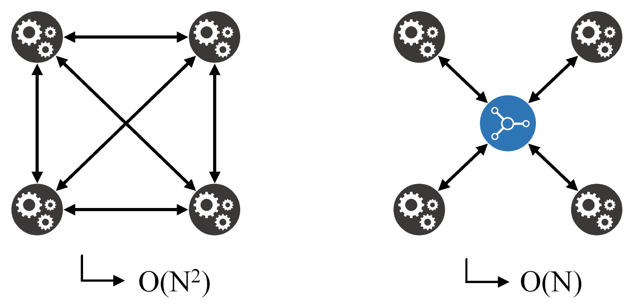

Figure 3. Interface complexity: Peer-to-peer vs. message broker topology

Figure 3 compares the complexity of a peer-to-peer topology to a message broker topology. Even if not all services have a connection to all other services, the cost factor of the interfaces is much higher with a peer-to-peer topology than with a centralized broker topology.

The breakeven for an interface concept using a message broker compared one with peer-to-peer interfaces is reached with a service number of $N = 3$ already. If there are more services than the use of a broker is in favor.

Using the message broker was especially helpful during the initial development, when the set of services changed constantly as the overall system grows. The development speed for implementing new features and adding new solvers did not slow down even when the number of logical interfaces increased. Even unstable services during rapid prototyping periods and testing did not compromise the availability of the message bus. Furthermore, Pulsar supports the deployment as container and in case of this study it means running in the same network environment as the attached services.

B. Protocol Buffers

In the previous work published in [2], the main messaging exchange format was JSON to be sent via HTTP Representational State Transfer (REST). Each service that sent data defined the schema of the data itself and the receiving services needed to look up that data schema and implement the data deserialization. One lesson learned from that work is that using such a decentralized message definition paradigm will rapidly lead to high efforts to maintain consistency between the implementation of the interfaces in the different services.

With Protocol Buffers a centralized message definition was introduced that could be easily controlled. Protocol Buffers support a wide range of libraries and compilers for various development environments to enable a transparent cross-platform interoperability. For the message instantiation in code itself, JSON provides an equal level of human readability, but when it comes to message schema definition, Protocol Buffers is in favor regarding human readability. A side effect of Protocol Buffers is the much reduced footprint of the binary data format on the network.

```
// JSON
JSONArray modelParameters = manifest.getJSONArray(↩
     JSON_KEY_PARAMETERS);
modelParameters.forEach(parameter -> {
    JSONObject manifestParameter = (JSONObject) parameter;
    if (manifestParameter.has(JSON_KEY_REFERENCE)) {
        [...]
    }
}

// Protobuf
List<FieldSpec> modelParameters = manifest.↩
     getInputSpecList();
modelParameters.forEach(parameter -> {
    if (parameter.hasReference()) {
        [...]
    }
}
```

These Java-snippets show on one limited example the difference in traversing through a JSON and a ProtoBuf datastructure. While the JSON part requires two string constants and a downcast, the ProtoBuf part does not have any of them. If the structure of the data is changed, the JSON part still is valid code, but will throw a JSONFormatException or ClassCastException at runtime. The ProtoBuf part will no longer compile and the compiler will show the programmer exactly where the problem is. This example shows the maintainability advantages ProtoBuf provides.

C. Service deployment using Docker

The different functionalities of the MBSE automation approach are realized by various services using different languages. In this case the layered structure of docker images shows its strengths, such as:

- Common functionality, e.g., corporate certificates, proxy settings, platform configuration files, are provided as base images.
- The storage requirements of derived service specific images consist only of the additional data added to the base image.
- For deploying new versions of services, only the changed layers need updates. This drastically reduces the network footprint when pushing large images to the registry and makes services available faster.
- With a CI/CD system in place that manages image dependencies, changes to base images can be automatically distributed to all derived services.

Some fundamental images that provide, e.g., a Maven, Python or npm environment are retrieved from public container registries. Sensitive derivations from these base images that contain proprietary corporate data, can be stored in a secured environment.

Part of the evaluation of new methods for MBSE automation was the ability to scale up to process large analysis tasks. Using Docker in a swarm mode enabled the distribution of the computational load across different Docker hosts. This is required when several tasks of the same kind, e.g., for hundreds or thousands of Design of Experiments (DoE) runs, have to be executed in parallel or when the structure of the PAM allows parallel execution of different tasks as shown by Figure 4.

More complex services, e.g., services that consist internally of a number of different individual containers, have been deployed as a stack. This eases the configuration of the

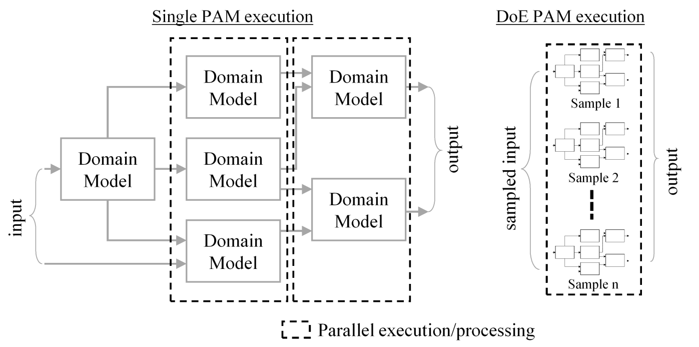

Figure 4. Parallel Model Execution

inter-service communication across the swarm via overlay networks which is managed by the swarm infrastructure itself. Whole service aggregations can be started and stopped by executing a single command or via a single click on the swarm management Graphical User Interface (GUI). Overall, this allowed deploying more services with a reduced functional complexity instead of having growing services that accumulate functionality to a level of unmanageability.

V. Outlook and future challenges

The main focus of this paper is on developing an architecture topology, applying CI/CD methodologies and using technologies that allow a more efficient service deployment, interaction and coupling. So far, there has not been a focus on the automation of the infrastructure deployment. An automated deployment not only of the services but also of the service infrastructure provides the capability to release different evolutions of the service implementations. It will also allow local replication of services, e.g., in case a company operates worldwide. There are a lot of challenges ahead to automate the release of multiple container environments under corporate restrictions, e.g., limited subnets, internet access restrictions, corporate certificates.

Furthermore, replicating environments tie a lot of computational resources. This is especially important nowadays that the CO_2 footprint of a company comes into public focus. Therefore, in a next step also different hardware platforms with a higher efficiency on power consumption, e.g., ARM-based servers, will be investigated. To succeed, measures need to be in place to have services that can run on such a hybrid infrastructure and can scale according to the computational requests even across those different hardware architectures.

VI. Conclusion

In a previous project published in [2], an approach for distributed computations in a heterogeneous tool landscape was already implemented. With an increasing service number it became more challenging to maintain the system as it was architected then. The extended analyses capabilities resulted in an increase in the number of services due to new and more processors for the problem evaluation. Also, the message content and structure became more exhaustive and complex. The approach required fundamental improvements to address new challenges such as the transition to a SysML-based modelling approach, more complex and larger system descriptions and a higher number of users and therefore increased computational load.

This paper shows how these challenges lead to fundamental changes in the new architecture. The introduction of a message bus with a publish/subscribe pattern makes a huge difference regarding implementation efficiency, maintainability and horizontal scaling. It also provided more system stability and availability.

Furthermore, the transition from a locally defined JSON based message encoding via HTTP REST towards a centrally defined message schema definition based on Protocol Buffers in conjunction with the message bus reduced interface inconsistencies between services drastically. It is now much easier to extend the software architecture with more services to solve even more exhaustive problem descriptions.

Code and Container templates helped to be more agile in a CI/CD environment. New functionalities were implemented and made available within hours. With the same development team and the former deployment paradigm as described in [2] such changes would have taken several days.

It can be concluded that the transition to a more structured approach for service interoperability can have a significant impact with respect to implementation time, efforts and resource costs.

References

[1] A. Maheshwari, “Industrial adoption of model-based systems engineering: Challenges and strategies,” Ph.D. dissertation, Purdue University, 2015.

[2] P. Helle, S. Richter, G. Schramm, and A. Zindel, “Coping with Technological Diversity by Mixing Different Architecture and Deployment Paradigms,” International Journal On Advances in Software, vol. 13, no. 1&2, 2020, pp. 92–103.

[3] P. Helle, G. Schramm, S. Klostermann, and S. Feo-Arenis, “Enabling multidisciplinary-analysis of SysML models in a heterogeneous tool landscape using Parametric Analysis Models,” 2022, submitted for The Complex Systems Design & Management conference (CSD&M 2022).

[4] N. Dragoni et al., Microservices: Yesterday, Today, and Tomorrow. Cham: Springer International Publishing, 2017, pp. 195–216. [Online]. Available: https://doi.org/10.1007/978-3-319-67425-4_12

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

[6] C. Currier, “Protocol buffers,” in Mobile Forensics–The File Format Handbook. Springer, 2022, pp. 223–260.

[7] P. Smet, B. Dhoedt, and P. Simoens, “Docker layer placement for on-demand provisioning of services on edge clouds,” vol. 15, no. 3, pp. 1161–1174.

[8] Docker Inc. Docker Overview - Docker Architecture. [Retrieved: July 2022]. [Online]. Available: https://docs.docker.com/get-started/overview/#docker-architecture

[9] Y. Pan, I. Chen, F. Brasileiro, G. Jayaputera, and R. Sinnott, “A performance comparison of cloud-based container orchestration tools,” in Proceedings of the IEEE International Conference on Big Knowledge (ICBK), 2019, pp. 191–198.

[10] H. Nurwarsito and V. B. Sejahtera, “Implementation of Dynamic Web Server Based on Operating System-Level Virtualization using Docker Stack,” in Proceedings of the 12th International Conference on Information Technology and Electrical Engineering (ICITEE), 2020, pp. 33–38.

[11] M. N. Mohd Mydin, B. Ismail, K. Rajendar, H. Ahmad, and F. Khalid, “An operational view into docker registry with scalability, access control and image assessment,” in Proceedings of the 7th International Conference on Engineering Technologies and Applied Sciences (ICETAS). IEEE, pp. 45–50.