



universität
wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„A Reference Model for Continuous Delivery for
Microservice Architectures“

verfasst von / submitted by

Michael Bültmann, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of
Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2018 / Vienna 2018

Studienkennzahl lt. Studienblatt /
degree programme code as it appears on
the student record sheet:

A 066 926

Studienrichtung lt. Studienblatt /
degree programme as it appears on
the student record sheet:

Masterstudium Wirtschaftsinformatik

Betreut von / Supervisor:

Mag. Dr. Hans-Georg Fill, Privatdoz.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Vienna, _____
Date

Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wien, am _____
Datum

Unterschrift

Abstract

Modern Information Communication Technology-devices are already widespread in the everyday life. With their communication ability they can create large networks exchanging information independently. The industry integrates these characteristics to their factories to support the interaction between the machines. The new approach helps to handle the complex scenarios and the fast changes in their demands.

The software development faces the same kind of challenges in its implementation and deployment processes of applications. Current systems are mostly complex, static and difficult to deploy.

An architectural approach, which supports the flexibility and the decrease of complexity of a software system, is the microservice architecture. Well-known companies like Amazon and Netflix introduced this pattern. Instead of a monolith system - a big coherent software unit with many dependencies - the microservices are small independent services with only one task. They are flexible in their creation and are able to be replaced without big efforts. There are also challenges with their integration, which are discussed in this thesis.

The automation of the deployment can be achieved through Continuous Delivery. The build of the software including all steps will be performed more frequently by the independent Continuous Delivery pipeline. This ensures less complicated roll-outs and faster time to market. Environments with both technologies, Continuous Delivery and microservices, are not often mentioned in research publications. Therefore the goal is to combine these technologies and create a reference model for the integration of Continuous Delivery in a microservice architecture. For this reason this work also provides a small overview of reference modelling in general. Afterwards the resulting reference model is presented and evaluated in an adaptation. In the end the results are discussed and an outlook is given.

Abstrakt

Informations- und Kommunikations Technologien sind bereits weit verbreitet im Alltag. Mit ihrer Fähigkeit zu kommunizieren sind sie in der Lage große Netzwerke zu erstellen, welche Informationen eigenständig austauschen. Die Industrie integriert diese Eigenschaften in ihre Fabriken, um die Kommunikation zwischen den Maschinen zu verbessern. Dieser neue Ansatz hilft dabei mit den komplexen Szenarien und die wechselnd Anforderungen umzugehen.

Die Softwareentwicklung stellt sich ebenfalls diesen Herausforderungen bei der Entwicklung und Auslieferung ihrer Anwendungen. Aktuelle Systeme sind meistens komplex, statisch und schwierig in Betrieb zu bringen.

Ein Architekturansatz, welcher der Flexibilität hilft und die Komplexität der Software erleichtert, ist Microservice-Architektur. Viele bekannte Unternehmen wie Amazon und Netflix haben dieses Muster eingeführt. Statt eines Monolithen - eine zusammenhängende Softwareeinheit - setzen Microservices auf kleine Services mit einer einzigen Aufgabe. Diese sind flexibel erstellt und sind ohne große Aufwände zu ersetzen. Bei der Einführung gibt es Herausforderungen, welche in dieser Arbeit diskutiert werden. Automatisierung der Auslieferung kann durch Continuous Delivery erreicht werden. Der Build der Software wird oftmals von der unabhängigen Continuous Delivery Pipeline ausgeführt. Dies führt zu vereinfachten Auslieferungen und schnelleren Inbetriebnahme. Umgebungen mit beiden Technologien, Continuous Delivery und Microservices, sind nicht oft Bestandteil von wissenschaftlichen Arbeiten. Daher ist das Ziel, die beiden Technologien zu kombinieren und ein Referenzmodell für die Integration von Continuous Delivery in eine Microservice-Architektur zu erstellen. Hierfür bietet diese Arbeit einen Überblick über die Referenzmodellierung. Anschließend wird das resultierende Referenzmodell vorgestellt und in einer Anwendung ausgewertet. Die Ergebnisse werden am Ende der Arbeit diskutiert und ein Ausblick wird gegeben.

Contents

Abstract	iii
1 Introduction	1
1.1 Smart networks	1
1.1.1 Cyber-physical systems	2
1.1.2 Internet of Things	6
1.2 Industry 4.0	8
1.2.1 Software production line	10
1.2.2 Comparison of the product manufacturing and the software development technologies	12
1.3 Research questions	13
2 Theory	17
2.1 Reference modelling	17
2.2 Microservices	21
2.2.1 Concept	22
2.2.2 Challenges	23
2.2.3 Advantages	26
2.3 Continuous Delivery	28
2.3.1 Pipeline	28
2.3.2 Characteristics	32
2.3.3 Case studies	35
2.3.4 Business, Development and Operations	39
2.3.5 Conclusion	40
3 Reference model	43
3.1 Planning	44
3.1.1 Planning of the software architecture	44
3.1.2 Planning of the infrastructure	49

Contents

3.1.3	Organization	52
3.2	Preparation	53
3.2.1	Preparation of the software architecture	53
3.2.2	Preparation of the Infrastructure	60
3.3	Configuration	62
3.3.1	Configuration of the software architecture	62
3.3.2	Configuration of the infrastructure	66
3.4	Reference architecture and infrastructure	67
4	Evaluation	71
4.1	Open Models Laboratory	71
4.2	OMiLab: Evaluation of the reference model	73
4.2.1	Planning	74
4.2.2	Preparation	78
4.2.3	Configuration	80
4.3	Results	96
5	Discussion	99
6	Outlook	103
	Bibliography	107

List of Figures

1.1	The concept of Cyber-Physical Systems	3
1.2	Framework of Cyber-Physical Systems: Conceptual Model . .	5
1.3	Framework of Cyber-Physical Systems: Key elements	6
2.1	eTOM: Business Process Framework Level 1	20
2.2	Scaling: Comparison between monoliths and microservices .	27
2.3	The Continuous Delivery Pipeline	29
2.4	Terms: Continuous Integration/Deployment/Delivery	31
2.5	The Deployment Process	39
2.6	Continuous *: Business, Development and Operations view .	40
3.1	Reference model: Overview	43
3.2	Planning of the software architecture	44
3.3	Planning: Split of microservices	45
3.4	Planning: Definitions of each microservice	46
3.5	Documentation of a template of a microservice	48
3.6	Planning of the infrastructure	49
3.7	Documentation template of the infrastructure	51
3.8	Planning of the organization	52
3.9	Preparation of the software architecture	53
3.10	Reference model: Development process	54
3.11	Preparation of the infrastructure	60
3.12	Configuration of the software	62
3.13	Dynamic infrastructure with Vagrant, Chef and Docker. . . .	64
3.14	Configuration of the infrastructure	66
3.15	Reference software architecture	69
3.16	Reference infrastructure	70
4.1	Components of a metamodeling method	72

List of Figures

4.2	Metamodelling based on language level	73
4.3	OMiLab architecture	74
4.4	Example of a documentation for a microservice	77
4.5	Example documentation of the infrastructure	78
4.6	POM.xml of the PSM service	79
4.7	OMiLab deployment overview	81
4.8	README.md of the GitHub Repository for the dynamic provision of the OMiLab environment	89
4.9	Jenkins: Configure job - General	91
4.10	Jenkins: Configure job - Source Code Management	91
4.11	Jenkins: Configure job - Build Triggers and Environment	92
4.12	Jenkins: Configure job - Build	93
4.13	Jenkins: Configure job - Post-build	94
4.14	Jenkins: Build Pipeline	95
4.15	Jenkins: Dashboard	95
4.16	Reference software architecture: Microservice architecture	96
4.17	Reference software architecture: Microservice	97
5.1	SWOT analysis of the reference model	100

1 Introduction

The evolution of machines has experienced a rapid development that is still on-going. For example the first locomotives used steam engines and have further evolved by integrating electrical energy and by introducing the digitalization. Today devices like smartphones are highly developed and are able to fulfill more complex tasks than in the past. The internet connection of these devices extends their range of possible applications and let them become "smart". They are used primarily to automate and simplify the tasks in the everyday life as much as possible (Lasi et al., 2014).

The software development has taken a similar approach when creating and introducing Continuous Delivery. It has been designed to support the developers at the automation of complex deployments. Through repeatable processes the delivery of software should be simplified. The different phases of the pipeline interact consistently with each other. This allows the developer to concentrate on the development. His reaction is only required if the pipeline fails.

The first section 1.1 focuses on the Internet of Things and cyber-physical systems. The following section 1.2 is about the term Industry 4.0 which relies on the concepts of the previous section. The last section 1.3 of this chapter handles the research questions and the transition to the software development area with their common approach of automation.

1.1 Smart networks

Before the 1970s most machines and devices like household devices had mechanical control units and no electronic parts integrated. After the launch of microprocessors it became more popular to use them in the appliances of

1 Introduction

the everyday life. Nowadays, embedded computers are widely spread and more powerful. A lot of them are used for example in cars and air-planes instead of the limited mechanical controllers. The next step is to connect these small computers and build a cyber-physical system (Wolf, 2009).

1.1.1 Cyber-physical systems

The embedded systems in the devices are capable of fulfilling many different kinds of tasks. There are sensor, controlling and measure units. Most of the time they work on their own and have no direct connection to other parts. Cyber-physical systems are looking for the potential that lies in their cooperative work.

The sensors can be wireless and distributed over a wide area. Through the communication with each other the devices are able to gain information about the physical part of the system. The new technologies represent the cyber part in this term.

Figure 1.1 of Schmid (2014, p. 62) provides a good overview of the topic cyber-physical systems and how the different members of the system work together. The main goal is to gather and process all the information in the physical system and make them easy to understand for the responsible operators. Through communication, controlling and computation the information are retrieved and handled.



Figure 1.1: The concept of Cyber-Physical Systems (Schmid, 2014, p. 62)

Rajkumar et al. (2010, p. 731) present a comprehensive definition for cyber-physical systems: "Such systems that bridge the cyber-world of computing and communications with the physical world are referred to as cyber-physical systems. Cyber-physical systems are physical and engineered systems whose operations are monitored, coordinated, controlled and integrated by computing and communication core."

Many domains including medical, aerospace, transportation and environmental are using devices like sensors to process their large amount of information. Well known examples of cyber-physical systems are smart factory or smart grid.

There are two factors influencing the development of cyber-physical systems: Trends of the application that are pushing for new features and needs which are directly pulled by the cyber-physical systems. The new technologies pushing cyber-physical systems are developments in the manufacturing of embedded computers. Devices can be produced at lower cost and their size decreases which extends their range of usage. With better power management, wireless receivers and internet bandwidth they can be applied to more scenarios.

Many areas need further development of the cyber-physical system technologies, for example healthcare, factory automation and process control.

1 Introduction

The upcoming trends and needs help to gain more control of the physical parts by using the new technologies (Rajkumar et al., 2010).

Rajkumar et al. (2010) show four examples of a cyber-physical system to clarify the upcoming challenges. The first example is an electronic power grid. Cities depend on the electric network and the power grid must be robust and reliable. Failures of the network can seriously impact the whole system and even cause total blackouts. The operators need a system to monitor and manage the power grid to prevent these faults. With the help of smart objects that are distributed in the network they are able to react quickly to occurring difficulties. Priorities of the power grid are resilience, security and reliability.

The second example targets the environmental problems the world is facing today. It summarizes several ideas. All the systems depend on the deployment of sensors in the nature which should observe the problem domain.

What impacts the melting of the ice-caps have, can be registered with a network of sensors that is easy to build and to maintain. This kind of information could be helpful to find and develop solutions. Food production is another field where cyber-physical systems can provide important improvements. Sensors can alert the workers if the temperature gets too high or if the water level in the ground is too low for optimal harvesting.

The third example is about the proper planning of evacuation at disaster scenarios like hurricanes. One goal is an efficient routing of transportation in a large-scale stressed situation.

The last example describes assistive devices. The number of people older than 65 years is increasing and these devices should help them in their everyday life. The tasks include cleaning, cooking and medical support. The helpers work semi-autonomous or fully autonomous. The challenges in this domain are trust, risk of failure, reliability, privacy and the right timing of actions.

These examples give a good overview of the range of possibilities a cyber-physical systems can provide and how the physical components interact with the cyber layer to process the information and make them understandable for involved people (Rajkumar et al., 2010).

1.1 Smart networks

The National Institute of Standards and Technology¹ (NIST) founded a working group in 2014 to standardize cyber-physical systems. This group created a framework that has been released in 2016. The working group consists of five teams for each aspect of the framework: Vocabulary and Reference Architecture, Cybersecurity and Privacy, Timing and Synchronization, Data Interoperability and Use Cases. The goal is to provide a guide for the creation of new and the maintenance/verification of existing cyber-physical systems.

The two figures 1.2 and 1.3 give a good summary of the framework.

The first figure 1.2 shows the conceptual model of the CPS framework. The approach starts considering a device being part of a system. This results in a system-of-systems. Each layer allows interactions with the human to provide information (Griffor et al., 2017).

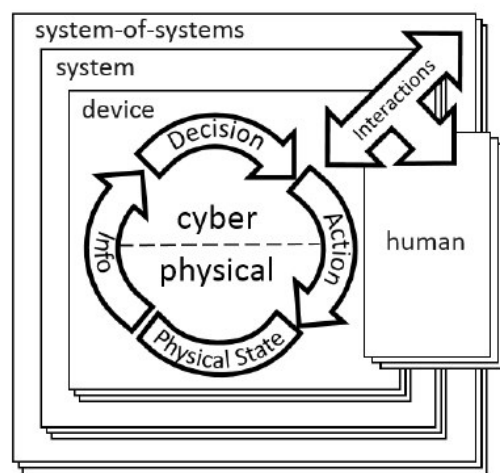


Figure 1.2: Framework of Cyber-Physical Systems: Conceptual Model (Griffor et al., 2017, p. 6)

The second figure 1.3 describes the key elements of a CPS. It consists of domains, aspects and facets. The domain is the environment in which the system-of-systems operates. The aspects consider all parts of the cps from a functional part to its lifecycle. The facets target the time-line of a cps from

¹<https://pages.nist.gov/cpspwg> (Accessed on: 05.12.2017)

1 Introduction

the conceptualization through the realization to the assurance (Griffor et al., 2017).

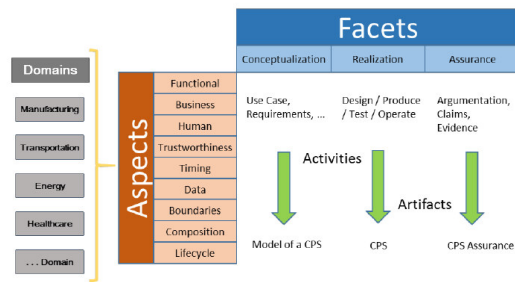


Figure 1.3: Framework of Cyber-Physical Systems: Key elements (Griffor et al., 2017, p. 14)

1.1.2 Internet of Things

The Internet of Things has a couple of aspects in common with the previously described cyber-physical systems. It can be defined as another term further to cyber-physical systems and sensor networks. The next step in the context of cyber-physical systems is to bring them to internet and focus more on the second level, the cyber level.

The Web 2.0 has provided the technologies allowing the user to contribute content to the web and to generate the wisdom of the crowd (O'reilly, 2007). The Internet of Things continues to develop these technologies further by enabling computers and devices to communicate with each other on their own and automatically. The following quote gives a good first definition of the Internet of Things: "The basic idea of the Internet of Things is that virtually every physical thing in this world can also become a computer that is connected to the internet" (Fleisch, 2010, p. 127).

Fleisch (2010) states that the objects do not become computers but that they have integrated small low-end computers. With the capability of tiny computers the objects evolve to smart things. As already mentioned in a previous section the smart thing relies on the development of the hardware

1.1 Smart networks

which includes smaller sizes, energy efficient devices and less manufacturing costs.

Furthermore, Fleisch (2010) lists some characteristics differentiating the objects of the Internet of Things from the ones of the internet. The first aspect is the appearance of the small things. They are mostly invisible and have only one task to fulfil, for example sensing and communicating. These devices are no full functional computers and usually do not have any interaction with humans.

So far there have been servers, computers and mobile devices in the internet and now the smart things expand it. The amount of network nodes increases in the Internet of Things and reaches the number of 84 billion nodes. Two additional characteristics target for the bandwidth and addressing of the smart things. A great number of smart things do not have a high bandwidth which is most of the time not needed, because they send only small packages of information. With the high number of nodes the addressing of all nodes is a big challenge. The next aspect is that the Internet of Things is more focused on the interaction of the machines among themselves and not on the exchange of information with the users. This corresponds to the last mentioned characteristic of foregrounding the sensing rather than the communication the internet is providing. The main task of the Internet of Things is sensing whereas the main task of the internet is to supply a way of communication. (Fleisch, 2010)

The next quote explains the Internet of Things related to the effects of the Web 2.0: "The Internet of Things adds another data dimension. It allows the physical world, things and places, to generate data automatically. In fact the Internet of Things is all about sensing the physical world" (Fleisch, 2010, p. 130).

This statement shows the upcoming challenges in a good way. The success will heavily depend on the approaches of how to handle the identification of nodes, the last mile communication and the resulting reachability of all smart things (Fleisch, 2010).

Some smart objects are only passive, because they only provide information for the identification of objects or products. Active objects can react, calculate or communicate with others. If products and objects would be equipped with the Radio Frequency Identification (RFID) they could become part of the business process. It is possible to read information from RFID and also write data to it (Botthof and Bovenschulte, 2009).

1 Introduction

Botthof and Bovenschulte (2009) have the opinion that the Internet of Things becomes more present in the everyday life and therefore present four different domains of the Internet of Things: logistics, production, healthcare and mobility.

Regarding logistics packages should be marked with a RFID and not with a barcode anymore. A barcode is limited with its information. Now it is possible that the package finds the way to the recipient on its own because all information needed for the routing is stored directly on the chip of the package. The importance of automation is increasing because the manufacturing of products is more and more divided into several processes taking place at different locations.

The production domain deals with the smart factory and will be covered in the next section about Industry 4.0.

Healthcare has already been mentioned in the section cyber-physical systems and addresses assistive devices.

Mobility targets the public transportation: vehicles should be able to communicate with each other and the remote maintenance of production systems should be controlled from far away.

Botthof and Bovenschulte (2009) also mention an interesting new requirement for the workers being responsible of monitoring the smart objects. Most of the time it is a simple task to observe the system, but when a problem occurs, they need to act quick and handle the complex situation with the necessary knowledge and experience. Another challenge is the privacy of user data. If products with a smart device are purchased by customers it should not be possible to derive buying behaviour or other information down to a single person.

1.2 Industry 4.0

The Internet of Things and cyber-physical systems lead to the collective term Industry 4.0 in the production domain. Connected networks and devices communicating with each other open the opportunity for the industry to implement them into their manufacturing facilities and to join the transition to a more digital world. All of the machines should have integrated chips or

sensors. With these smart wireless objects they can exchange information during the manufacturing process, and it can be easily monitored by the responsables. The goal is to be flexible with the planned tasks and alignments of machines by producing a customized unit in a mass production environment. At the beginning of the production each product gets an ID which stores the information and requirements for this specific unit, so it can control its own process. Then every machine in the production chain knows what to do with this particular product. In order to create an efficient configuration of the work flow it is recommended to organize every single station of the pipeline in a modular way. Thus, the machines will know what to do with the product at every station. Furthermore it is possible to rearrange the station order if necessary.

This highly-connected factory has many advantages and is discussed under the term of Industry 4.0. The name refers to the version numbers of a software program. Industry 4.0 is described as the fourth evolution of the industry. After the mechanisation, the introduction of electricity and digitization to the manufacturing the next step is the integration of Information Communication Technology (ICT) devices in the production chain (Lasi et al., 2014). As a collective term *Industry 4.0* also represents concepts such as smart factory and has a large impact on the working life of humans in modern factories.

The transition to a smart factory follows the organizational and structural process changes resulting in advantages and as well as challenges.

The first advantage of a factory which is characterized and influenced by Industry 4.0 are shorter times of development and adaptation. Single machines can be reconstructed easier and faster. This is an important benefit that helps to shorten the time-to-market.

The individualization has been already mentioned and allows the production to meet the requirements of single customers. The range of possible configurations increases the flexibility significantly. Further, the resources are used more efficiently and decisions can be made faster.

The development potential of technologies is similar to the cyber-physical system. For the future the focus is to increase the automation and the networking of sensors and to decrease of the size of the smart objects. The challenges are to dispose of an effective way of planning and organizing the new system. The integration and transition from existing factories to a more

1 Introduction

digital company is a complex task which needs to be addressed accordingly. One of the research domains is the creation of a modelling method and reference models to represent smart factories (Lasi et al., 2014).

Kagermann et al. (2012) mention that the introduction of cyber-physical systems connecting the physical and cyber world requires the provision of a common ground for the technical processes and their corresponding business processes. This occurs in a smart factory of the production domain.

The organizational structure of the companies changes and Botthof (2015) for example discusses the impact on the working life of the employees. There are more flat hierarchies. In smart factories there will still be workers but their range of tasks will differ in the future. More skilled people will be needed to maintain the machines. The automation will reduce the responsibility of the individual worker and his control possibilities. The number of employees in a company might also change. It can increase or decrease, all depending on the working area and its development. Further, Industry 4.0 will face challenges mainly regarding the working life. These are legal issues concerning privacy and liability, standardization and IT-security.

1.2.1 Software production line

In the beginning products were handmade and every resulting good was an individual one. When the production line was introduced by Ford Motors, more people were able to afford standardized products which were easier to produce and therefore less expensive. But it was not possible to add individual demands of the customer any more. The variety was reduced. These two types of products can also be adapted to the software engineering. There is individual software which is made for the specific needs of the customer but the implementation is expensive. On the other side there is standard software that is affordable, but does not have all the features the customer requires (Pohl, Böckle, and Linden, 2005).

Pohl, Böckle, and Linden (2005) connected and adapted the production line of the car industry to the software engineering domain. The next step is mass customization. There should be mass production of goods with consid-

ered customer requirements. This represents an important core aspect of the approach of Industry 4.0. The car industry has created common platforms for its cars so that parts of a vehicle can be used in other vehicles as often as possible. This increases again variety and reduces cost. The sales also went up because the customers were able to choose from different options. The solution results in two aspects, platforms and mass customization. The adaptation of the two principles to the software domain is the main goal of Pohl, Böckle, and Linden (2005).

When preparing a production line for multiple products, there must be a common platform including all of them. This is essential for the mass customization later on. Creating this platform and its components is almost the same procedure as the decision of software engineering on how to split and define the borders of modules and microservices. *Which parts of the software can be reused and integrated in a different application?* The parts of the software which are highly customizable must be addressed accordingly in order to separate them from the stable code for example.

When defining the parts of the platform their flexibility must be considered particularly. It must be possible to easily reuse them in other scenarios. A platform can be assigned to one product line. It is recommended to share and spread the platform features to other platforms to create groups with similar characteristics. By making the platforms available to the outside the all product family platforms will profit from each other.

The organization of the company is also affected by the change to platforms and new organizational units must be defined. This is also quite similar to the consequences of integrating Continuous Delivery and microservices to the organizational structure which will be discussed in following chapters. Cost reduction is drastic once multiple varieties of products have been introduced. Newly added products benefit from the existing platforms. Only for a small number of products it is not effective to create a platform. The break-even-point is at about three to four products.

In the software development it is also efficient to reuse existing modules and services as often as possible. Also infrastructure technologies which have been introduced with Continuous Delivery can be reused for multiple applications such as the build server.

One positive effect of reusing the parts of a platform is the improvement of quality: the parts are used more frequently and therefore faults are faster

1 Introduction

identified. This also results in better maintenance. After the installation of the platform or platform groups the release of new products is much easier and the time to market much shorter. It has a similar effect in the software engineering, when a new application can rely on existing modules and only a small amount of new codes has to be implemented. In this case it is also easier to calculate the costs for developing the new application (Pohl, Böckle, and Linden, 2005).

In the end Pohl, Böckle, and Linden (2005) have introduced the resulting term Software Product Line Engineering which is based on platforms and customization for software development. A software platform should consist of interfaces with a common structure. For this common structure it is necessary to define technologies and process flows to ensure the re-usability. The key principles are re-usability, flexibility and variability that will be covered in the following chapter about new technologies, Continuous Delivery and microservice architecture.

1.2.2 Comparison of the product manufacturing and the software development technologies

Table 1.1 shows a comparison of the characteristics of the manufacture of products in the Industry 4.0 domain and applications in the software development with focus on the new technologies Continuous Delivery and microservices. Both try to deal with the rising complexity of automation and separation in small units.

A manufactured product can be related to an application of software development. This is the result of the creation process. The automation of creating a build of an application with continuous integration has similarities to the pipeline of machines in a factory as it is completely automated. The modular alignment of machines refers to the microservice architecture. Different alignment or combination of machines or respectively services can lead to new products and applications.

1.3 Research questions

Industry 4.0 / Production line	Software Development
Manufactured product	Application/program
Alignment and use of machines	Programming language, pattern and architectural structure
Automation of manufacturing	Continuous Delivery
Machine pipeline	Continuous Delivery pipeline
Modular structure of the machines	Microservice architecture (independent services)
Production of single customized unit	Easy deployment for each scenario
Common platform for different products	Infrastructure for continuous delivery
Organisation changes by platform	Organization changes by teams
Easy to add similar products	Easy to create new applications with existing services
Smart object has only one task	A microservice has only one task

Table 1.1: Compare Internet of Things and Software Development

Each smart object in the grid has only one task with its sensor or other functionality. A microservice should also have only one purpose. The common platform which helps to produce single customized units relates to the required infrastructure of Continuous Delivery, which can be reused for several projects in different environments.

1.3 Research questions

The software development also targets to automate processes as much as possible and to let the different stages of the implementation process interact with each other. Some aspects of the Industry 4.0 can be transformed. Continuous Delivery (CD) became popular in the research field after companies have used this approach to deliver their software more efficiently. CD is an automated pipeline which builds, tests and deploys the software. Another new concept is the fragmentation of the functionality of software

1 Introduction

into small units. This is a step further than modularization and is called microservice. Previously it was common to build and deploy one big coherent unit of software. In the end it was very difficult to maintain and to add new features, because changes affected always the whole software. Even if it was built and divided into several modules, there would be many dependencies between the different parts. Microservices are small strict independent units and should be deployed separately. Every microservice has its own purpose and not more.

In this work I want to handle three research questions regarding existing environments of CD with microservices, a reference model for the integration of CD to microservice architecture and a practical adaptation of the reference model.

The **research questions** are the following:

1. **Which conceptual, organizational and technical challenges exist with CD and how can these approaches be handled with the addition of microservice architecture?**
2. **How can a reference model be defined for the integration of Continuous Delivery in an architecture with microservices?**
3. **How does the reference model apply to an existing scenario?**

The first research question *"Which conceptual, organizational and technical challenges exist with CD and how can these approaches be handled with the addition of microservice architecture?"* targets the existing environments. The process of achieving an insight into the current approaches is a literature research on both topics: microservices and Continuous Delivery. The answer of the research is an overview of the characteristics and a current status of working implementations. The results are shown in chapter 2.

For the integration of CD and microservices there are only a few public documentations available. In my work I want to provide the answer to the second research question *How can a reference model be defined for the integration of continuous delivery in an architecture with microservices?*

The results of the first research question help with the derivation of the reference model. The findings will be analyzed and combined. The reference model covers the integration of both techniques, CD and microservices, and is presented in the chapter 3.

1.3 Research questions

In the end it is necessary to verify if the created reference model can be applied to a real life example: *How does the reference model apply to an existing scenario?*

For achieving this I apply the reference model to the microservice-based architecture of the website of the research laboratory OMiLab at the University of Vienna. The basis of the adaptation is the resulting reference model of the second research question. Afterwards the results will be evaluated and are documented in chapter 4.

The two technologies and the reference modelling are explained in chapter 2 and are the basis for the following chapter. Chapter 3 presents the resulting reference model of the second research question. The chapter 4 is about the third research question and includes the evaluation of the reference model. The conclusive chapters target the Discussion 5 and the Outlook 6.

2 Theory

In this chapter the first section 2.1 contains an introduction to reference modelling. This should provide basic knowledge for the upcoming reference model in the following chapter. The remaining two sections handle microservices (2.2) with their concept and characteristics and the Continuous Delivery (2.3) approach.

2.1 Reference modelling

Vom Brocke (2015) starts the definition of reference modelling with the term of a model by Stachowiak (1973) and its three aspects. A model represents something that already exists and abstracts it. The way of the abstraction is the last and important aspect of the definition of models. The abstraction focuses on a specific feature or characteristic of the original model. A map of a subway network for example represents the subway network itself in an abstracted way by only focusing on the connections between the stations and not on the correct geographical location of the stations. Modelling describes the creation of models. Reference models are models, which can be reused or are specifically created to be reused. The process is called reference modelling (Vom Brocke, 2015).

Fettke and Loos (2005) describe reference modelling as all actions of the construction and the usage of reusable models. Reference models should help to adapt to specific models from a universal model. The main objective is the reusability of the models. According to Fettke and Loos (2005) there are two main types of reference models, models of the subject area and the propositional logic area. In the subject area the reference models are derived from the reality.

2 Theory

Reference models based on the propositional logic are developed by scientists and their research results. There are more possible interpretations of reference models. These consider interpretations of a group of terms in a domain, abstraction of a general class, useful techniques or normative actions.

In the context of reference modelling there are the reference modelling language and the reference modelling method. The reference modelling language defines the representation and the reference modelling method gives guidance on how to model the resulting reference models. No standard modelling language has been defined so far. The variety of different domains leads to multiple language groups. The languages are natural or formal. Common languages are the semi-formal language event-driven process chain (EPC) and entity-relationship models (ERM). The standardization of reference modelling languages is a big challenge for the future. Other areas of future discussions are the increase of different domains, the size of models, the construction and the evaluation. (Fettke and Loos, 2005)

In Fettke and Loos (2004) the authors discuss the process of the construction and the usage of reference models in another publication. Both processes have four steps trying to ensure the quality of the used reference model. The construction process consists of the definition of the problem, the development, the evaluation and the maintenance. The usage process passes the following steps: selection, adaptation, integration and usage. The successful execution of the construction and usage process can guarantee a high quality of the resulting reference model.

Popular examples of reference models are:

- Open Systems Interconnection (OSI) model
- Supply Chain Operations Reference (SCOR) model
- enhanced Telecom Operations Map (eTOM)

Open Systems Interconnection (OSI) model is published by the International Organization of Standardization (ISO)¹ and organizes the network communication in multiple layers. The first layer addresses the communication of single bits, the seventh and last layer the application the user works with.

¹<https://www.iso.org> (Accessed on: 14.01.2018)

2.1 Reference modelling

Each layer defines the usage of protocols, units and devices.

The Supply-Chain-Operations-Reference (SCOR) model² is a reference model created by the non-profit Supply-Chain Council³ to help companies with the definition of their business processes and common view on the supply chain. Therefore it is widely known and accepted.

The SCOR model consists of four layers and every layer describes the processes more in detail. The first layer divides the processes in categories defined as plan, source, make, deliver, return and enable. The second layer groups the processes in different types like planning, execution and enabling. The third layer consists of the detailed processes and the fourth layer is the company-specific instantiation of the processes.

The enhanced Telecom Operations Map (eTOM) provides a framework of processes and data models to manage the services of telecommunication companies. It handles all processes of the provider, which are offered by the network to the customers.

Figure 2.1 shows the first level of the framework⁴ with the categories that have some similarities regarding the structure and categories to the SCOR model.

²<http://www.apics.org/apics-for-business/products-and-services/apics-scc-frameworks> (Accessed on: 14.01.2018)

³<http://www.supply-chain.org> (Accessed on: 14.01.2018)

⁴<http://casewise.tmforum.org/evolve/statics/frameworkx/index.html> (Accessed on 14.01.2018)

2 Theory

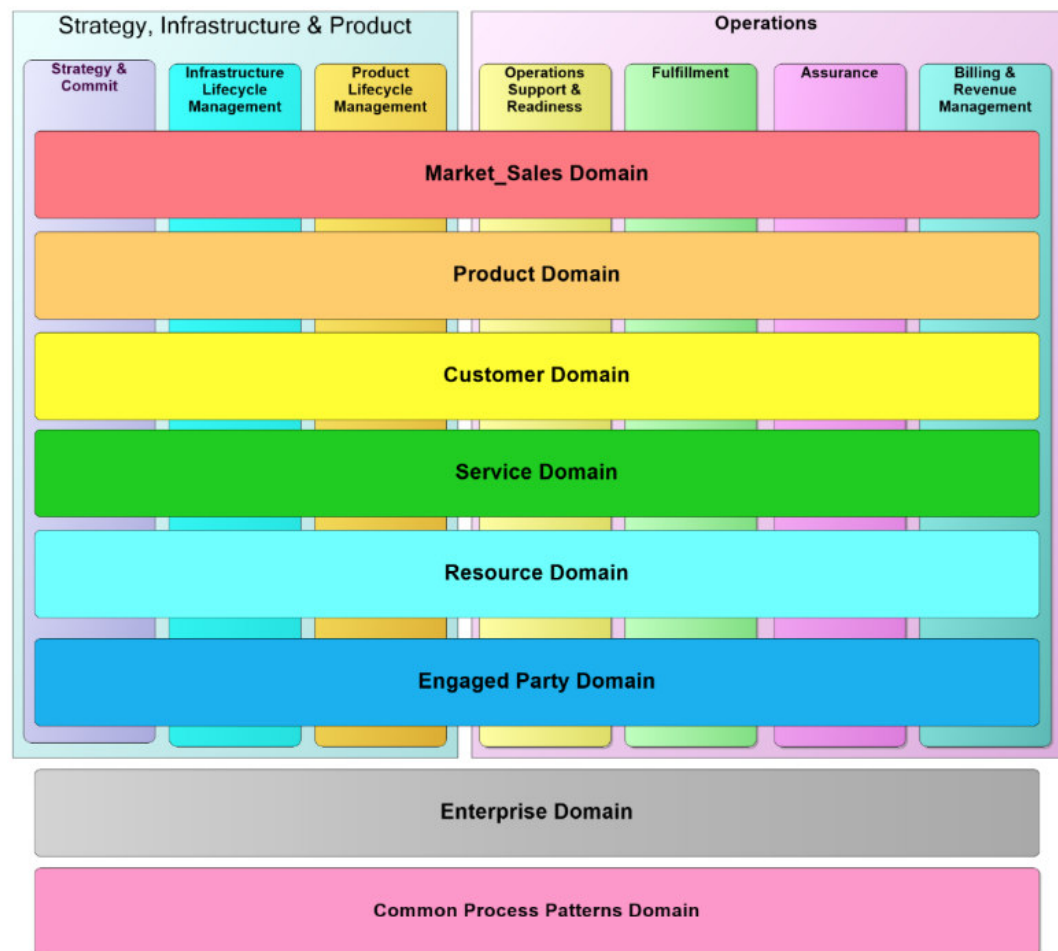


Figure 2.1: eTOM: Business Process Framework Level 1 (4)

The framework is organized as a matrix. The domains of product, customer and service are horizontally aligned. Strategy, infrastructure and product, as well as the operations are placed vertically.

2.2 Microservices

As already mentioned in section 1.2.2 there are similarities in the concepts of Internet of Things and new technologies in the software development. The first term of this transition is microservice and is discussed in this section. Namiot and Sneps-Sneppé (2014, p. 24) consider microservice architecture as a suitable approach to Internet of Things: "We think that the unified (monolithic) framework for M2M (IoT) is not a realistic solution. By this reason we think that microservices are the natural fit for M2M (IoT) development."

Microservices are independent services that deploy on their own.

"A microservice is a minimal independent process interacting via messages." (Dragoni et al., 2016, p. 2)

The communication is mostly handled through calls over the network. A system depending solely on the usage of microservices is a microservice architecture.

"A microservice architecture is a distributed application where all its modules are microservices." (Dragoni et al., 2016, p. 2)

In the software development it has been common to build and deploy one big coherent unit of software. This approach is a monolithic architecture and the opposite of the newer microservice architecture.

In a monolithic architecture the complexity increases rapidly by updating to newer versions. In the end it is very difficult to maintain and add new features, because changes affect the whole software. Even if it is built and divided into several modules, there would be a lot of dependencies between the parts. For the developers it is a challenging task to understand the software with all its components. The coordination complicates the process and the productivity decreases. The scaling and chosen technology is not as flexible as developers would like them to be. Microservices help to solve these problems. Microservice architecture is the next step from service-oriented architecture to more independent components.

At first, microservices were discovered by and introduced to the industry.

2 Theory

It became more popular and found its way towards the research area. Nowadays research papers about microservice architecture are published more frequently (Thönes, 2015).

The sections of this chapter analyze microservices in more details. The first 2.2.1 section describes the general concept of microservices. Afterwards the section 2.2.2 lists the challenges which can arise during the integration process. The last point shows the advantages and the opportunities of integrating a microservice architecture.

2.2.1 Concept

A microservice should have only one goal and purpose to fulfil. There is a clear responsibility of the single service. Due to its small size and work capacity should be easily understood. The most important feature is the independence of the other services. A microservice should be able to perform on its own. Deployment, scaling and tests should be separated from other services. Due to the small size and one goal to fulfil, a microservice is easy to replace by another one which is written in a different programming language. (Thönes, 2015)

The popularity of microservices has grown over the past years because of the possibility to migrate from monolithic systems to a microservice architecture. The problems occurring at monolithic systems have been already mentioned earlier. Microservices allow the developers to change single parts of the system step by step to independent services. Monolithic systems are very difficult to maintain after a long period of continuous changes. Now they can split up features of the monolith and implement a microservice to replace the part of the application instead. With the separation into services it is possible to scale better and to use the advantages of continuous delivery for the single independent service. Multiple microservices are easier to organize and maintain regarding further new change requests than a monolith. (Thönes, 2015)

The most common use-case of introducing microservice architecture is to build it by starting from a monolith. Usually companies already have large systems and want to migrate microservices. Implementing a microservice architecture on a greenfield is possible but not often used. Examples for

2.2 Microservices

moving from monoliths to microservices are Netflix and Amazon (Thönes, 2015).

The communication pattern is based on a choreography approach with intelligent endpoints and dumb pipes. The services should act smart and handle all incoming and outgoing messages on their own. There is no centralized managing tool for the communication nor a message routing system. The business logic is in the endpoints and not in the pipes or the service bus. RESTful-protocols are preferably selected and based on the HTTP-protocol. This is the same basis as the world wide web (Fowler and Lewis, 2015).

Microservice architecture evolved from service-oriented architecture (SOA). One of the main aspects of service-oriented architecture is the modularization of the system. The focus is to separate the services more from each other and have independent services. Systems which are based on service-oriented architecture are mostly organized as an orchestration. There is one unit being responsible for the management of the whole system. Microservice architecture tries to use choreography because of the self-contained services which should work on their own. It is also possible to design one microservice as the managing unit and use orchestration. A microservice architecture prefers to build on choreography as the main communication pattern (Dragoni et al., 2016).

Fowler and Lewis (2015) describe the architecture style of microservices as a system which consists of a set of smaller services.

2.2.2 Challenges

One of the biggest challenges of a microservice architecture is the creation of a distributed system which increases the complexity of the system. With new applications and tools to manage the system it becomes easier to control the whole system but there is still some manageable overhead to monitor. The complexity depends largely on the fact that there are independent services with more remote calls over the network than in-process function calls. The remote calls are mostly asynchronous and can result into delays (Namiot and Sneps-Sneppe, 2014).

2 Theory

The complexity which rises when building and working on a monolith will be shifted to the network layer in a microservice architecture. With cloud services and applications to manage infrastructure of the new network complexity is easier to manage and monitor than in the past (Thönes, 2015).

Usually systems are divided in components which use in-memory calls to communicate with each other. Microservices on the other hand use web-service calls because the separation takes place in independent services and not in components. The calls over the web need more time than the in-process calls because the services are organized in a distributed architecture. (Fowler and Lewis, 2015)

A system with several microservices becomes more difficult to perform tests in the environment. Dependencies between services even increase the efforts of testing (Namiot and Sneps-Sneppe, 2014).

The decision where the services should be split goes along with the size of the microservice. This is a tough question. There are different strategies like split by use-case, verbs or entities of the functionality of the service (Namiot and Sneps-Sneppe, 2014).

A good approach for splitting into services is an important aspect. The resulting architecture should have as few as possible dependencies of each other. (Fowler and Lewis, 2015)

One of the most difficult challenges for developers working with microservices is the handling of database access. There are several approaches to be followed. It is possible that every service has its own database and inconsistencies will be handled afterwards. This allows faster response times. Another approach is to dedicate to the database a service for the access. This service provides an API for accessing the database and can add restrictions for usage (Fowler and Lewis, 2015).

Developers need also to meet the requirement for the deployment of several services. The deployment of several services has to be managed and the necessary infrastructure must be provided. Continuous Delivery and dynamic provision solve the large part of deployment (Namiot and Sneps-Sneppe,

2014).

The following chapter will point to this more in detail.

Microservices can take over parts of a monolith that need to be changed and reimplemented. When starting from a greenfield it can happen that a system which was designed as microservice has many dependencies and results in a monolith. A good approach is to start with a monolith and to do a refactoring of the code. The transition is done step by step and the API of the monolith is used while developing the microservices. There can be problems in the transformation regarding the splitting of the microservices and the responsibilities of different layers (Fowler and Lewis, 2015).

The companies structure their teams regarding to specific abilities. Most of the time there are teams in the development of user-interfaces, backend and databases. These layers of teams reflect to the software architecture. Each team will develop components representing their team goals. Changes to the system often affect multiple layers and this can cause complex efforts.

This behaviour was discovered by Conway and is known as Conway's law:

"Any organization that designs a system will produce a design whose structure is a copy of organization's communication structure." (Conway, 1968, p. 1)

The organization of a microservice architecture wants to break with the law of Conway and structures their teams by services and not by set of skills. A team being responsible for a microservice should consist of experts of every layer and thus be able to develop a service on their own. The organization of the teams is not project-related any more. A team supports the whole life cycle of the service. It is responsible for any changes and for the deployment at the end. In the past the developers were only responsible for the implementation and not for the support or the deployment (Fowler and Lewis, 2015).

Another challenge is the awareness of the team of the new technology and the resulting organizational changes. The members of the team must accept the new approach of developing software (Fowler and Lewis, 2015).

2 Theory

2.2.3 Advantages

As previously mentioned microservices should be comprehensive and perform just one task. The purpose of the service is as small as possible. There is no limitation of the size or the lines of the code. It can vary from a few lines to a couple of hundreds. The developers should keep in mind that if the service gets too large, it is probably a good idea to split the service into two services. The service should fulfil only one task well (Thönes, 2015). The microservice architecture allows to use different programming languages. Independent services only interact over the APIs respectively the network layer and do not do direct function calls to other services. The main focus is to keep the services light weighted by avoiding any overload of features in one service. The developers can decide which programming language they want to use and which one suits well for the given problem (Thönes, 2015).

Besides the selection of the programming language and the responsibility of a team for the whole service, the decentralization and the independent structure of microservices require ways of securing the quality of the service. Teams follow service contracts and with a consumer-driven contract they get feedback from the users in a short time cycle (Fowler and Lewis, 2015). A microservice architecture is well designed for ongoing changes because only one microservice needs to be modified. The behaviour of the changes influences the way of splitting the microservices. Stable code and parts with many changes are divided into different microservices if it is possible. This guarantees good interchangeability and extensibility. Compared to the microservice architecture it is more complex to change parts in a monolith because of coherent components (Fowler and Lewis, 2015).

Due to the individual deployment of each service it is possible that a microservice will no longer be available. Some companies perform specific stress tests to monitor the reliability of the services and of the entire system. It is important to observe how the system reacts if one service is not reachable any more. The desired goal is to provide a high tolerance and resilience. On the other hand the operating staff must be aware of failures in the system. For managing the service it is necessary that the system has a good monitoring and logging component. A microservice must log reasonable information to find the errors easily when they appear. The monitoring

2.2 Microservices

service should provide every information like status and workload to the developers on a centralized space. Monoliths can be structured similarly but the difference is that a monolith is deployed in an environment and has always access to all components. (Fowler and Lewis, 2015)

Another advantage resulting from the single deployment is the scaling of services. In combination with a load balancer it is possible to deploy multiple instances of a microservice. Figure 2.2 shows an example of how the scaling of microservices works (Fowler and Lewis, 2014).

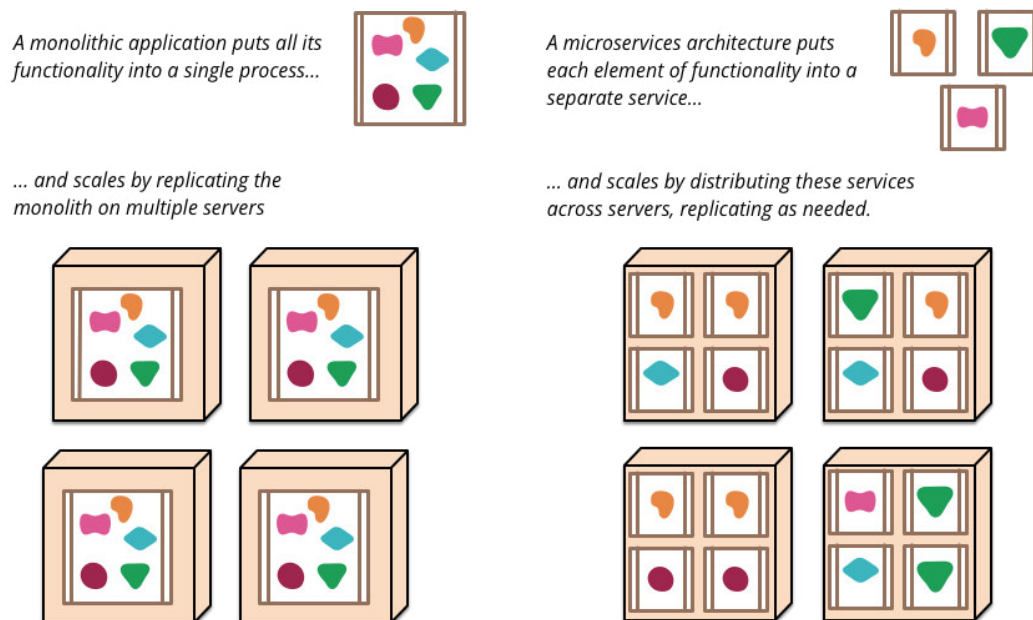


Figure 2.2: Scaling: Comparison between monoliths and microservices (Fowler and Lewis, 2014)

An interesting aspect that can occur with a set of services is the composition of them. The developers can configure and put together the service they need for their system. A company could produce different types of applications and assemble a software product by selecting from a pool of already existing microservices. An important precondition for realiz-

2 Theory

ing this is to have implemented a good way of splitting the microservices. Every microservice should fulfil a specific purpose (Fowler and Lewis, 2015).

The communication patterns like message bus and publish & subscribe fit well in a microservice architecture. Services can be split and organized similarly to the parts of a smart network (Namiot and Sneps-Sneppe, 2014).

2.3 Continuous Delivery

One of the biggest problems of monolith architectures is the deployment. Integration of a new release to a productive environment can lead to a lot of difficulties. Before the deployment the codes of all involved teams have to be tested and built afterwards. This process is complex and not rarely without the occurrence of errors. Continuous Delivery (CD) is a good approach to avoid errors in the deployment process. CD is a deployment pipeline with the main steps commit, build, test and deploy. After each commit to the code repository there will be automated tests and on-demand roll-outs for testing or productive environment. CD can be very efficient if microservices are used because only single microservices go through these pipeline steps and not all parts of the whole system. CD can lead to improved productivity, faster building times and better quality (Chen, 2015a).

The first section starts with the description of the pipeline. The second section shows the characteristics of Continuous Delivery with challenges and advantages. The following two sections target case studies of existing scenarios of companies and the terms Business, Development & Operations. The last section contains a conclusion.

2.3.1 Pipeline

The pipeline usually consists of three main phases. The following sections provide a short description of each phase. Figure 2.3 shows how the phases are put in order.

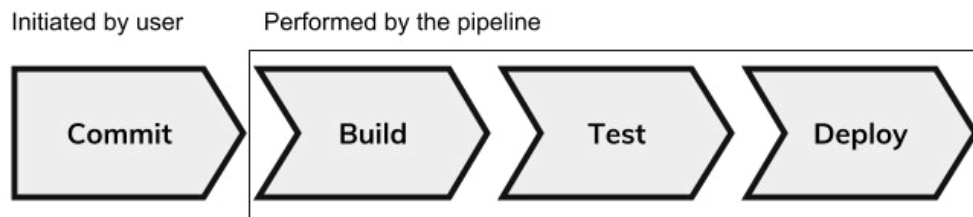


Figure 2.3: The Continuous Delivery Pipeline

A more technical view on the pipeline will be presented in the following chapter 3 about the reference model.

Commit

Before the Continuous Delivery can perform the first run, the developers need to commit the new features to the source code repository. The commits have to be executed on a regular basis and more frequently than in former approaches. With more commits the amount of code in the change decreases and errors are easier to detect. Also hidden features to be activated in future releases should be included in the commits. Commits should consist of a meaningful message with a change request number of the issue tracking system and description. It is also helpful to mark the code with comments when there are specific changes (Wolff, 2014).

Build

The first step being executed by the continuous delivery pipeline is the build. Depending on the programming language the sources will be compiled and a package is built. The build also includes the unit test to ensure its functionality. It is the first phase where the pipeline can fail and the developers need to react quickly to fix the issue. There should always be a deployable build (Chen, 2015a).

2 Theory

The unit test is an important part of this phase. It covers the whole code and is the basis for a good quality of the application. It tests functions and methods for the right and expected return values (Phillips et al., 2014).

The results of this step of the pipeline are the artefacts or binaries of the software which can be deployed later to the selected environment (Chen, 2015a).

Test

The test phase includes more complex test suites. There are different kinds of tests that can be performed.

For example performance test analyses the resilience of the system. *How does the application handle a large amount of concurrent users? Can the system manage the high server load and what happens when a service is not reachable anymore?* Some companies want to stress their system so much that instances of services break down and they can see how the system handles the upcoming load.

Parts of acceptance tests can also be automated. Frameworks of acceptance tests can interact with web forms and automatically assign inputs to the form elements. Depending on the input the framework can check if the resulting page is correctly displayed (Chen, 2015a) (Wolff, 2014).

At the end there is the possibility of manual tests. If a test environment is available, there is the opportunity to do a manual test before the deploying reaches the production status.

Steinacker (2016) presents an interesting approach to the manual test. The continuous delivery pipeline contains a plugin which takes screenshots and then compares them with the previous version. If changes in the view occur the pipeline will advise to perform manual tests.

Deploy

At the end of the pipeline is the deployment. There are different approaches to and terms for the deployment.

2.3 Continuous Delivery

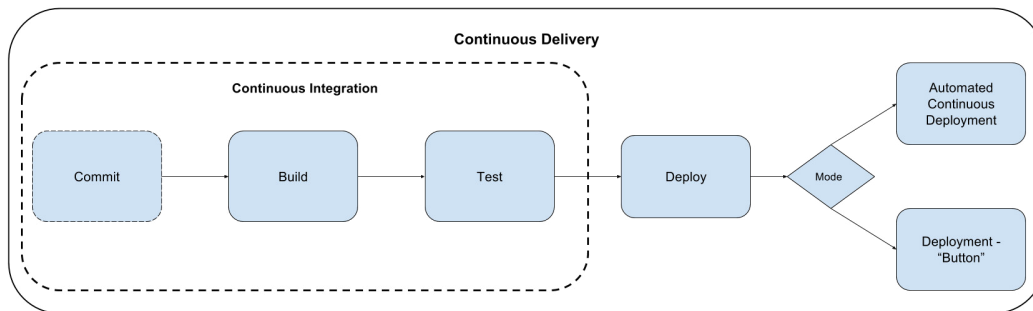


Figure 2.4: Terms: Continuous Integration/Deployment/Delivery

Figure 2.4 shows the three different terms of continuous integration, delivery and deployment. Continuous integration handles the on-going build of artefacts. Continuous deployment describes the whole execution of the continuous delivery pipeline with all phases including a direct deployment to the customer's environment. Continuous delivery is the generic term (Wolff, 2014).

Besides continuous deployment there are more alternatives to deploy the software with the pipeline to the productive environment. With a test environment new releases can be tested and after a successful manual test the approved version can be deployed to the public environment. It is also possible to perform a blue/green deployment with two environments. One scenario represents the current state of the software and the other one the previous version. For each new deployment the older environment will be used for the update and afterwards switched to the production. This ensures the availability, because there is no downtime and the possibility to easily switch back to the previous version.

Another opportunity for the deployment is a canary release. If a system consists of several instances behind a load balancer, one of the instances will be updated by a new release. By monitoring or restriction of certain test users the developers can control the behaviour of the system. After successful tests more instances will be updated (Wolff, 2014).

It is also possible to introduce a gatekeeper to the deployment process. The gatekeeper controls the deployments to the production environment. For

2 Theory

example it allows to deploy to production only in a certain timeframe or other restriction. This approach weakens the continuous deployment but declines the risk of failures (Steinacker, 2016).

2.3.2 Characteristics

Continuous delivery provides an effective approach of deployment which is fast and reliable. In most companies the deployment of new releases, especially for a monolith architecture, is time-consuming and bears a high risk of errors due to manual actions. Since there are many sources for mistakes, releases are rarely scheduled and not often practiced. Continuous delivery uses shorter release cycles with an automated deployment. Everything should be automated and under version control (Chen, 2015a).

Advantages

In consequence of the shorter release cycles new features can be published and launched sooner. Thus, the users do not have to wait until the full release cycle will be completed and can benefit much earlier from the new features. Their feedback to the developers takes less time and allows faster reaction to it.

The time between development and feedback being much shorter the developers still have a quite fresh memory of the implementation. In the end the product suits better to the expectations and the developers can react faster to requirements for modifications. Also the planning does not have to be as strict and fix as before thanks to the flexible release cycles (Chen, 2015a).

As previously mentioned many efforts are needed for the setup of an environment. Continuous Delivery builds and deploys a test environment automatically. Here is a large potential of saving time and increasing productivity. After the successful deployment in a test environment the release of the new application is easy. A lot of manual tasks become obsolete because of being automated and the number of activities with no added values but causing unnecessary efforts will be considerably reduced (Chen, 2015a).

Another advantage resulting from shorter release cycles is the reliability of the release, because any changes between two versions are not extensive. If

2.3 Continuous Delivery

problems occur it will also be easier to roll back to previous version compared to a monolith. The reliability is ensured by the repeatedly processed tests. If an error is detected the pipeline will stop immediately and notify the developer. This behaviour avoids the occurrence of bugs and the time for debugging decreases significantly. A high level of reliability guarantees a good quality of the software product (Chen, 2015a).

The successful usage of Continuous Delivery can improve the motivation of the team and especially the trust of the customer in the application. Due to the new approach more users are satisfied with the results (Chen, 2015a).

The automated tests are necessary to avoid errors the user can experience. There is instant feedback from the pipeline when building and compiling the software (Phillips et al., 2014).

Challenges

Documentations about the introduction of Continuous Delivery are rarely spread. Analogical to the changes in the organization of a microservice architecture, Continuous Delivery also needs some adaptations in the organizational structure. New teams must be involved and work together. For example system administrators must provide access to the infrastructure during the deployment phase (Chen, 2015a).

The cooperation of the developers with the operation provides better control over the infrastructure and facilitates the creation and maintenance of the environments. Also, it has turned out to be very effective and useful to reproduce scenarios and simply change the environments by configurations (Phillips et al., 2014).

A complete solution that can be adapted to developers' needs is not available on the market yet. There is no standard solution so far. All tools must be put together and configured individually. This can be very time-consuming until there is a suitable and working pipeline for Continuous Delivery (Chen, 2015a).

The tools of the pipeline should provide the user with an interface to manage the pipeline. It is helpful to orchestrate the pipeline from one central point to obtain a clear and all-embracing overview of the whole process.

2 Theory

This is the opposite to microservice architectures where the services are organized as a choreography (Phillips et al., 2014).

The transition from formerly large complex releases to more small sets of new features is important. Deployments of massive releases can cause several problems in different environments. Also the build, which contains a lot of changes of several developers and is rarely performed before, can cause problems and can have a negative impact on quality. The new approach with Continuous Delivery requires to get every additional feature right one by one and to proceed only to the next feature once the build and the deployment work fine with the previous one (Phillips et al., 2014).

It is necessary for Continuous Delivery that the developers often perform commits to the code repository and that the integration of the code is processed repeatedly. The workers need to adapt to the new working pattern. Frequent commits allow numerous runs of the pipeline with build and tests contributing to high quality (Phillips et al., 2014).

The introduction of Continuous Delivery is time-consuming. It is not recommended to change everything at once right at the begin of the integration. The first objectives are the findings of bottlenecks and the improvement of the process at this point. After the first accomplishments it is a good idea to extend all possible automation processes. A slower approach will ensure a smooth implementation of Continuous Delivery and a by far easier handling (Phillips et al., 2014).

Chen (2015b) describes implications of the usage of Continuous Delivery and summarizes them under six architecturally significant requirements. The first requirement is the deployability. Deployments should be performed more regularly and frequently. Releases that take several months and result in a longer downtime are not acceptable anymore.

The more restarts and reboots of the software the more the infrastructure will be vulnerable to attacks. This implication should not be ignored.

Most of the time a new deployment replaces the old version and consequently loses the information of the old one after the transition to the new version. Therefore the third requirement addresses the logging of the deployments which should preferably be done by a logging server.

Due to the smaller release cycles and the faster feedback from the users the code will have to be modified more often. The developing team must keep in

mind the frequent modification of the code base while the implementation of new features is going on.

Similar to the previously mentioned microservices the monitoring of Continuous Delivery is very helpful. A centralized interface for the distributed pipeline is important to obtain an overview of the whole system. Instant feedback from the pipeline to the developers is of high importance.

The last requirement points to the test phase. The code goes through several tests with the aim to have the application ready for the release. The code must be tested extensively to provide the necessary level of quality. Tests must be created for every service to be efficient while working with Continuous Delivery (Chen, 2015b).

2.3.3 Case studies

Olsson, Alahyari, and Bosch (2012) present a “stairway to heaven” in their article which describes the evolution of how software companies are planning their product development. This stairway consists of five steps starting with the traditional development. This level is characterized by long development cycles prior to the delivery. On the second level the ways of development are more flexible and versatile. There is more interaction with the customer and the release cycles are getting shorter. The third level represents continuous integration. The success depends heavily on an increased cooperative work, frequent commits of all developers and automated builds. This is a prerequisite for further integration of Continuous Delivery. The next level is based on the previous one and is about continuous deployment. Deployments should be performed more often at the customer’s site resulting in better feedback and shorter release cycles. The last level shows an even faster reaction to customer requests and immediate deployment of the requests. It stands for the optimized scenario.

In their case study the authors observed four companies within the first three levels that have been close to the final stage. In their interviews they were asked how to achieve continuous deployment and how to handle customer feedback instantly. The goal was to identify the problems of the different companies and to provide first recommendations for solutions. The transition from traditional to agile development requires changes of the

2 Theory

management structure and a closer bound to the customer. The teams have to be organized in smaller units to better focus on single features instead of whole components.

The companies' main problem of introducing Continuous Delivery is to build test suites with tests and validations. Significant for using continuous integration is the regular commit of all developers to one branch and to split the demands in smaller parts.

The last transition the authors observed at one company was the change to continuous deployment. The problems are the handling of different customer configurations and the way product management communicates with the customer to provide feedback to the developers. The product management needs to adapt to the shorter release cycles and to manage the configurations in consultation with the customer (Olsson, Alahyari, and Bosch, 2012).

Leppänen et al. (2015) are studying the transition of 15 companies to continuous deployment in their research publication. The goals of the companies vary concerning the release cycles because of the different domains they operate in. The target deployment scenarios are fully automated continuous deployment, capable continuous deployment, on-demand deployment and calendar-based deployment. The companies want to improve the management of their releases. All companies look for a Continuous Delivery pipeline including the integration and the build of their application. Only a few wish to be able to deploy directly to the productive environment. There are different reasons and obstacles for not using a fully automated continuous deployment in these companies.

One reason against a fully automated pipeline is the resistance towards change. For introducing Continuous Delivery organizational changes are recommended such as to form new teams. Some companies do not intend to change their structure.

Another roadblock preventing the complete deployment to the productive environment are the preferences of the customer regarding the stability and privacy.

Barriers for the continuous deployment are not only caused by single customers but also by the working domains. In the health domain the stability and safety of the software are critical. Undesired and unforeseen side effects are not acceptable in a medical environment with patients. Manufacture

2.3 Continuous Delivery

related software faces difficulties, if a certain downtime is exceeded when deploying the new version. This can cause an important loss of venue.

Sometimes tests do not cover all possible error sources or the developers are not able to implement further tests. It happens also that developers are not willing to deliver broken builds to the customers because this bears the risk of lowering their trust in the software. For some scenarios the setup is too complex and requires too many efforts to get it to an automated stage.

Many customers are using legacy systems that are not easy to replace because they have grown over a long period of time. Deployments with an existing legacy system can be tricky and therefore another reason to be careful with continuous deployment.

A large code base of complex software can also cause some issues with continuous deployment. The time of build, tests and deploy can exceed an acceptable time span or the downtime of the system during the deployment is too long. Another obstacle mentioned by the authors is the manual test. Sometimes it is not possible to generate suitable tests with static test suites. This results in manual tests containing the risk for undetected errors. Also for security reasons the creation of test routines matching the requirements is difficult.

The last issue targets the different databases. Deployment in different environments affects the status of the database. This can lead to unexpected behaviours.

Depending of the technical requirements of implementing a pipeline and the preferences of the customer the fact to own a fully automated Continuous Delivery pipeline with continuous integration and continuous deployment can be very beneficial. Considering the previously mentioned obstacles the automated deployment should only be used under certain conditions (Leppänen et al., 2015).

Claps, Svensson, and Aurum (2015) are investigating how the company "Atlassian" moved the development of their products to continuous deployment. They use interviews with the development staff for this purpose. The authors list technical and social challenges which have appeared during the process. Some of them have been already mentioned. They do not perform a complete continuous deployment at Atlassian, because there is still an additional manual step within the deployment to the customer's environment

2 Theory

in place.

Technical challenges are the dark features. If a feature is too large it will be divided into small batches. Then the parts are delivered step by step to the customer scenario and switched off till the feature is complete. The developers should implement new requirements in extra branches and merge them to the master branch during the development progress.

Most of the listed challenges are social challenges. The team members have the impression of being under consistent pressure for providing an updated state with all new features at any time.

If developers and respectively the management do not understand the principles of continuous deployment they will not recognize the benefits and will not have any motivation to move to continuous deployment.

The adaptation of Continuous Delivery has also a large impact on the existing roles within the team. The members have more responsibility for the deployment because it moved from the operation to the development. Now they are responsible for the whole distribution of the application. The tasks of product management and support hotline changed, too. In the past the marketing was applied to one major version and the reported bugs were assigned to a specific version. With Continuous Delivery the marketing and support changed, because now there are more releases and it is more difficult to refer to a major version.

Furthermore due to shorter release cycles and small change sets, bugs can be removed faster. Still, the occurrence of bugs being reported by a customer is strongly related to the quality of the automated tests. Especially for continuous deployment the developers must be aware of the relation between automated tests and quality. The last interesting challenge the authors discovered is the feature discovery. Since there are multiple releases within a short time and the customer is not directly informed about updates, it is possible that they do not notice new features. Sometimes new features the development team spent a lot of time for are not recognized by the customer. The authors recommend to publish any changes via a web-blog to keep the customer up-to-date.

Figure 2.5 presents a simple deployment process (Claps, Svensson, and Aurum, 2015).

2.3 Continuous Delivery



Figure 2.5: The Deployment Process. Derived by (Claps, Svensson, and Aurum, 2015)

2.3.4 Business, Development and Operations

This cooperative approach of the developers and product management extends the term DevOps to BizDevOps. Development and Operations strive to supply software in a fast and efficient way to the customer. The next step is to involve the business level for gathering relevant feedback from the customer.

Fitzgerald and Stol (2014) show the three phases of the software life-cycle in figure 2.6. It consists of business strategy, development and operations. The focus is on the links which are called BizDev between business strategy and development and DevOps between development and operations. All stages try to deliver the product as fast and as good as possible.

2 Theory

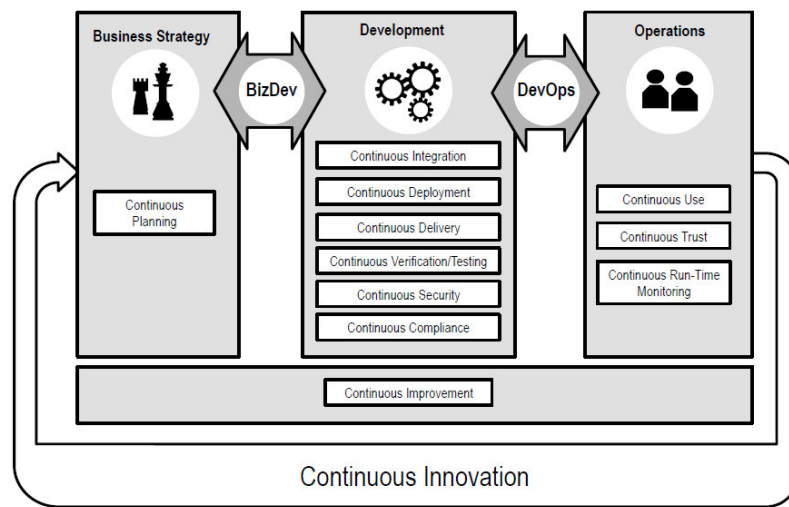


Figure 2.6: Continuous *: Business, Development and Operations view.(Fitzgerald and Stol, 2014)

In DevOps the development and IT operations work together to provide an efficient way of implementing the new software. The development implements the software and the operation supplies the necessary infrastructure. Continuous integration, continuous deployment and continuous delivery belong to a development including influences of the operation (Fitzgerald and Stol, 2014).

DevOps and Continuous Delivery can also be seen as a lean manufacturing derivative. Lean manufacturing focuses on the terms value and waste. Everything adding no value to the product is considered waste and should be removed. Another important aspect is that processes should be automated as far as possible. Automation ensures a high level of quality (Fitzgerald and Stol, 2014).

2.3.5 Conclusion

Continuous delivery fits well in a microservice architecture. Monoliths with a large code-base are difficult to bring into a continuous delivery scenario.

2.3 Continuous Delivery

Therefore monoliths are rarely delivered with various possible complications. Long compile or build times do not work well in a pipeline. It is easier to build a delivery pipeline for small independent services with a specific purpose like a microservice. A well working Continuous Delivery pipeline offers benefits such as shorter time to market and higher quality.

Challenges arise for both technologies, for example in the organization. In the microservice architecture the decision to split services and the responsibility for the whole life cycle of services affects the organization. The new development behaviour in a Continuous Delivery scenario with frequent commits and cooperation with the operations changes the organization.

The following chapter presents a reference model of the integration of Continuous Delivery to a microservice architecture. The chapter also provides information about the technical implementation of both technologies.

3 Reference model

Due to the fact that both techniques are relatively new, there are only a few documentations of the integration of continuous delivery with the transfer to a microservice architecture ((Wolff, 2014) & (Wolff, 2015)).

This chapter describes a reference model to integrate continuous delivery to a microservice architecture. The reference model also provides examples of the technical implementation.

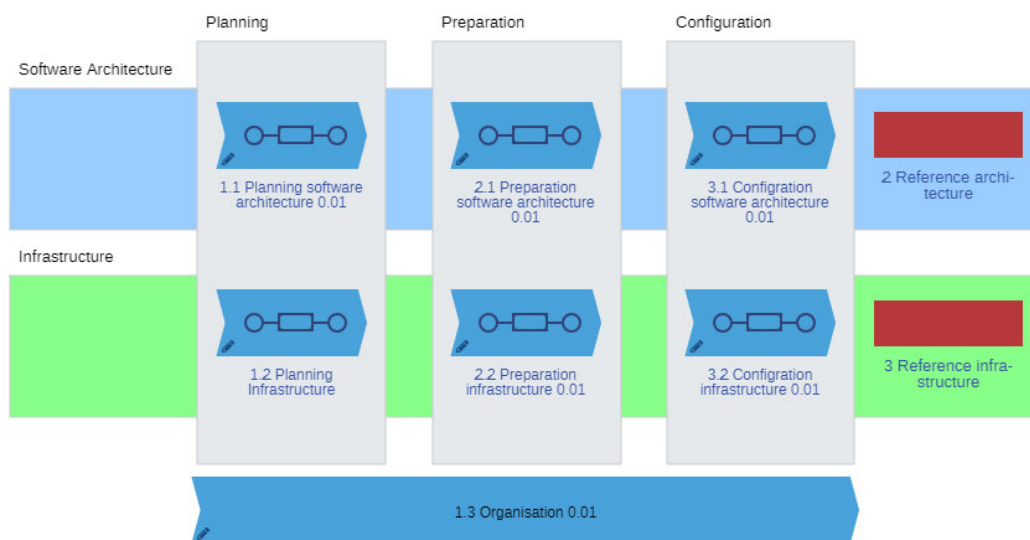


Figure 3.1: Reference model: Overview

The overview of the reference model in figure 3.1 shows the structure with the vertically aligned three phases planning (3.1), preparation (3.2) & configuration (3.3) that are explained in their respective sections.

The horizontal layers are the software architecture, the infrastructure and

3 Reference model

the organization. The organization is the basis for the reference model that is present in all phases with its required aspects.

Software architecture and infrastructure separate the three sections into two subsections for each phase.

The structure of this reference model relates to the previously presented examples of reference models in section 2.1.

3.1 Planning

In general, the planning has to consider all needed aspects prior to the start of the implementation.

In section 3.1.1 the goals of the software architecture layer are the definitions of the split of the microservices, the communication through interfaces and the selected programming language.

Section 3.1.2 targets the general requirement of the hardware for the infrastructure.

3.1.1 Planning of the software architecture

Figure 3.2 shows the process of the software architecture planning with every single task.

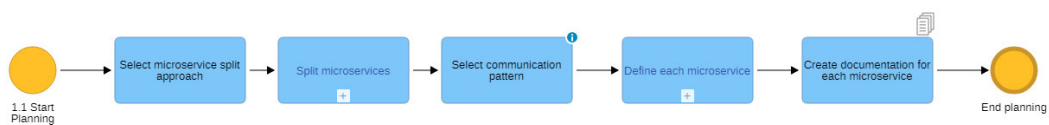


Figure 3.2: Planning of the software architecture

One of the most important tasks of the planning phase is the split of the microservices. This decision affects the organization of the team and the structure of the application equally.

There is no general approach regarding the split of services in a microservice

3.1 Planning

architecture as it varies from company and project.

As for the migration from a legacy system. In this case it has to be defined which components of the existing system need to be added or re-factored. This is a useful way of creating a microservice architecture by reducing the functionality of the legacy system sequentially.

For a new project it is helpful to define the different components of the application and their tasks. As for the components it should be discussed either to divide one service in more fine-grained services or to find mutual characteristics of several services to be covered in one. There might be possibilities to reuse a microservice in another component, in a future extension of the application or even in a new project (Wolff, 2015).

The simple approach of a split algorithm is shown in figure 3.3.

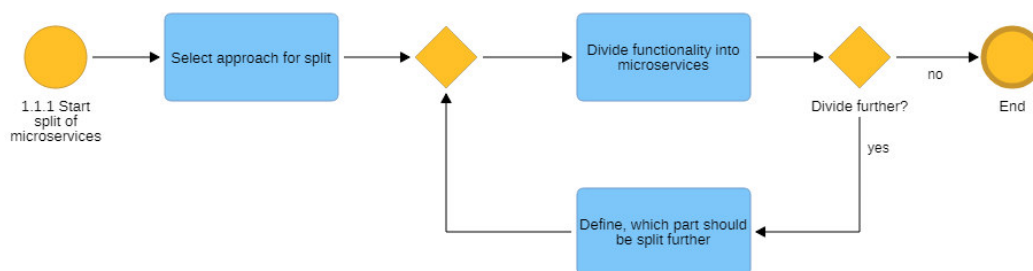


Figure 3.3: Planning: Split of microservices

The starting components of a web-shop for example are *product*, *customer* and *invoice*. Each microservice provides information through an interface to the others. Depending on the requirements the microservice *invoice* can be separated further into *order* and *shipment*. Customer relationship management might need to reuse the microservice *customer*, product management might find it important to reuse the microservice *product* including further specific microservices. This needs to be discussed with the developers and the later users.

After having defined the structure of the microservices the next step is to determine the handling of the communication between the services. Each microservice provides data in a specific format through its interface to the

3 Reference model

other services in the network. This must be documented and shared with all teams.

There are multiple ways and patterns of implementing the interfaces. Most common way is to provide a REST (Representational State Transfer)- API (Application Programming Interface) that works through the HyperText Transfer Protocol (HTTP) and returns the data in a JavaScript Object Notation (JSON) format. The API uses the method types GET, POST, PUT and DELETE of HTTP to modify the data for the specific URL (Uniform Resource Locator) path (e.g. "../customers/id"). Another output-format is the extensive markup language (XML). The implementation of a RESTful service is well supported in many programming languages. An alternative is Web Services Description Language (WSDL) which works with the HTTP or Simple Object Access Protocol (SOAP) and allows using Remote-Procedure-Calls¹ (RPC) (Mulligan and Gračanin, 2009).

The point-to-point connection pattern of REST endpoints is most frequently used. It is possible to integrate a pipeline coordinating the requests or to organize the communication through asynchronous messages².

The second to last task of the software architecture planning process concerns the definitions of each microservice and is documented in the sub-process 3.4.

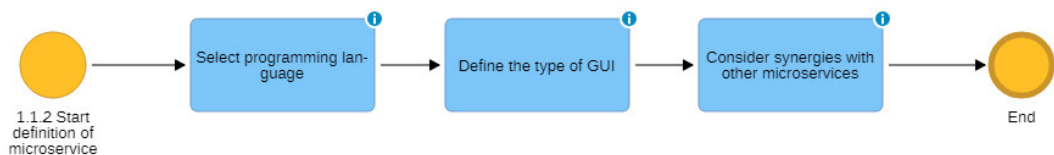


Figure 3.4: Planning: Definitions of each microservice

The first point of the sub-process is the selection of the programming language to be used. This correlates with the definition of the interface.

¹<https://smartbear.com/learn/api-design/what-are-microservices> (Accessed on: 15.10.2017)

²<https://capgemini.github.io/architecture/is-rest-best-microservices> (Accessed on: 15.10.2017)

3.1 Planning

There might be positive synergies when teams use the same programming language. However, in a microservice architecture it is not relevant for one team to know the programming language used by another team as long as their microservice provides the data in the defined and expected way. Depending on the programming language the build mechanism can change and they may not use the same build tool/server. This should not limit the teams at the selection of their programming language, but it needs to be considered.

Common programming languages for microservices are JAVA, Node.js or PHP with their respective frameworks.

The graphical user interface (GUI) must also be defined in this process. *Should the microservice provide a GUI or only the data to other services?* A good approach is to include always a low-level GUI for the other teams (Wolff, 2015).

The last task of the parent process is the creation of a documentation of each microservice. This documentation should contain all basic information, the responsible persons and the technologies and include all decisions that have been made so far. The documentation is important because it is shared with other teams to ensure having everyone on the same page. An example template could look like the document in figure 3.5.

3 Reference model

07.08.2018

Documentation



Microservice Documentation <Microservice Name>

General

Name:
Description:
Responsible:
Responsible (Deputy):

Technologies

Programming language:
Frameworks:
Build tool:

Interfaces

Interface	Description

Tests

Framework:
Description:

Deployment

Destination:
Description:

Figure 3.5: Documentation of a template of a microservice

If REST is applied the microservice needs to provide the endpoints. There

3.1 Planning

are tools available to help to document these easier. Swagger³ for example provides the functionality to generate a documentation of the REST-API. It lists all endpoints with their path, parameters and responses in a JSON or HTML-format and can be included for example in the build tool. Later on, the documentation of the methods is extended by annotations⁴.

3.1.2 Planning of the infrastructure

The process 3.6 documents the tasks when planning the infrastructure.

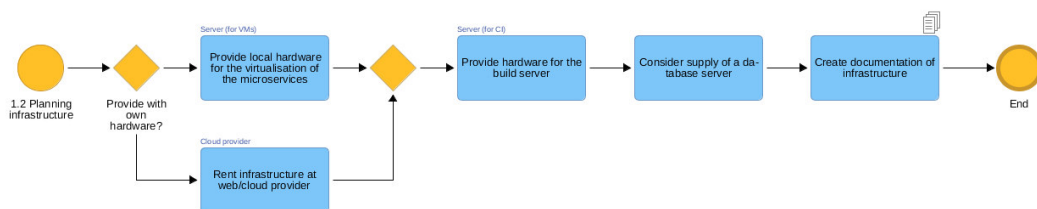


Figure 3.6: Planning of the infrastructure

At first, a decision must be made how the infrastructure should be provided. It is possible to choose an online provider or to host the hardware in the own company. If a mutual infrastructure is available and acceptable for the teams they can use the same infrastructure. In order to warranty the independence of every microservice each team has to have their own infrastructure if at all possible. Concerning the number of microservices and their estimated workload the operations must provide the necessary hardware resources to run the servers. Another option is to use the scaling of microservices and to create multiple instances of a service on demand. The type of delivery to the test and productive environments needs to be considered, too. Besides the utilization of local hardware there is the alternative to provide infrastructure by renting cloud/web hosting environments. It is necessary to evaluate

³<https://swagger.io> (Accessed 02.06.2018)

⁴<https://springframework.guru/spring-boot-restful-api-documentation-with-swagger-2> (Accessed on 02.06.2018)

3 Reference model

the provision of local servers with own hardware or online infrastructure requiring a certain fee.

Further to the hardware for the new application the infrastructure of Continuous Delivery requires certain components such as an independent server where to run the build server application. This server must fulfil the needs of the build server which in most cases is the presence of a web-container like Tomcat.

If there is no issue tracking system already in the company, it is recommended to integrate one and reserve the necessary resources.

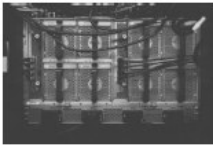
After the decision for the software architecture there is usually the need for a database instance. Depending on the split of microservices and how they should interact with the database, this can vary from one database giving access to only one dedicated or to all microservices up to multiple databases for each microservice.

This decision influences the hardware requirements and the scaling of the database instances.

Similar to the process of the software architecture the infrastructure also needs to be documented accordingly. A template of the documentation can look like the example in figure 3.7.

07.08.2018

Documentation



Infrastructure Documentation

Version Control System

Type / Technology:
Location:
Responsible:
Description:

Build server

Type:
Location:
Responsible:
Requirements:
Description:

Deployment

Type:
Locations:
Responsible:
Description:

Figure 3.7: Documentation template of the infrastructure

The documentation should contain the three major parts: version control system, build server and deployment. The mentioned components will be discussed in the following sections.

3 Reference model

3.1.3 Organization

The tasks of the organization have to be defined in the planning phase but they are valid throughout the whole integration process and establish the basis for the integration of continuous delivery and microservice architecture. The process is shown in figure 3.8.

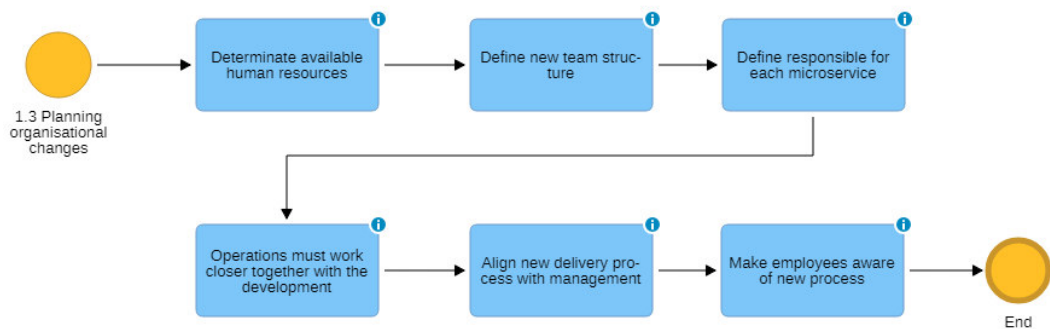


Figure 3.8: Planning of the organization

The organizational structure of the team is important for achieving a successful integration of microservices. The impact of the team structure on the structure of the applications has been explained in section 2.2.2.

All team members must be aware of the new process of development.

The teams have to be defined according to the needs of the project. The members should not be picked by their specialties but instead each team should for example consist of developers for both frontend and backend, as well as of deployment-experts. The responsibility of the whole life-cycle of the microservice including development, deployment and support should be assigned to the corresponding team that can be responsible for more than one microservice. If so, this should be recorded in the documentation of the microservice.

For microservices having more dependencies to each other than normal, it is recommended to assign the owner of the services to the same team. It could also make sense to group microservices with similar tasks to the same team. These decisions should be documented and accepted by the teams to minimize delays and to determine responsibilities in time.

3.2 Preparation

A presentation including detailed training and hands-on sessions for the developers are always useful and will, for sure, increase the awareness about the new procedures. Additional to the necessary changes to the development teams there are changes to the operations which need to work closer together with the development. This has been already mentioned in section 2.3.4.

The changes reach the management equally. Management needs to consider shorter and smaller release cycles and to provide feedback from the customer to the development much faster than before.

3.2 Preparation

The preparation phase is based on the decisions of the planning and continues by selecting of the version control system, the build tool/server and the test framework. The preconditions for the integration of the Continuous Delivery pipeline are handled in section 3.2.1. It refers partially to the following phases of the pipeline: commit, build, test & deploy.

Additional to the requirements of the software architecture there are also tasks for the infrastructure being described in section 3.2.2.

3.2.1 Preparation of the software architecture

Figure 3.9 presents all aspects that need to be considered during the preparation of the software architecture.

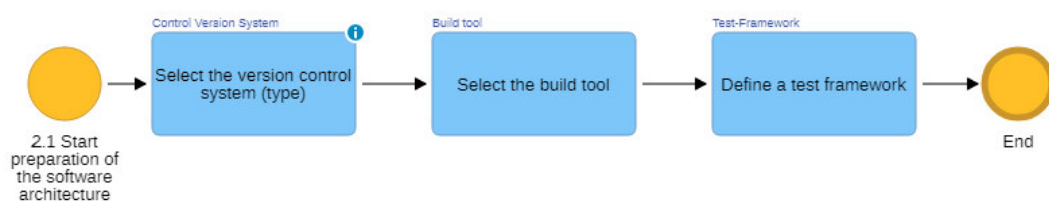


Figure 3.9: Preparation of the software architecture

3 Reference model

The selection of a version control system is the precondition for the commit step of the pipeline in 2.3.1.

A version control system (VCS) is required for the Continuous Delivery pipeline, in order to enable the build server to retrieve the sources from a main repository. This will be explained in detail in section 3.2.2. There are several version control systems available on the market. Nowadays Git⁵ is popular.

Figure 3.10 shows an example of how a common development process runs.

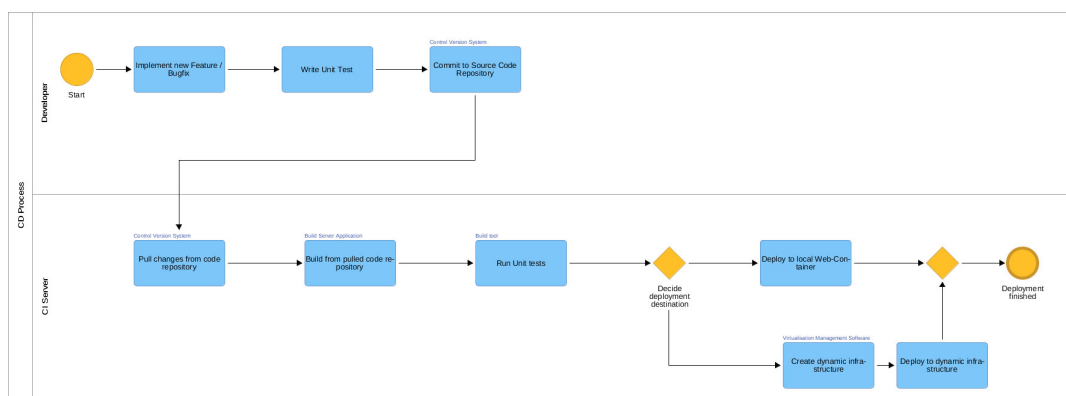


Figure 3.10: Reference model: Development process

Rowson (2017) presents a good overview of the current version control systems. In general there are two different types of version control systems, centralized version control systems (CVCS) and distributed version control systems (DVCS).

A CVCS stores the code repository at a central server which is accessible to all developers. Users can checkout projects from the server and commit their changes back. In a project with multiple developers there is potential for conflicts. When the same file is changed at the same time by two developers between commits, a developer has to decide manually which the right part is. Representatives of CVCS are Concurrent Versions System (CVS)⁶ and the further developed Apache Subversion (SVN)⁷. SVN was the first popular

⁵<https://git-scm.com> (Accessed on: 19.10.2017)

⁶<https://savannah.nongnu.org/projects/cvs> (Accessed on: 19.10.2017)

⁷<https://subversion.apache.org> (Accessed on: 19.10.2017)

3.2 Preparation

CVCS and is still one of the most used. It is supported of many developing tools.

The newer technology is DVCS where repositories are primarily local at the developer's workstations. The developer implements new features, he commits changes to his local repository and once a feature is completed he pushes the commits to the central repository at the server. Most common for this type of CVS is Git. Git is well-known from the open-source code repository platform GitHub⁸. Another representor of DVCS is Mercurial with Bitbucket⁹ as an example of source code hosting.

The structure of having local code repositories at the developers' and a global repository at the server facilitates more frequent commits which is recommended while using Continuous Delivery. Collecting local commits and pushing them altogether to the server instance reduces the appearance of conflicts (Rowson, 2017).

Brindescu et al. (2014) did a survey with developers about their experiences with version control systems. In their publication they observed code repositories of both types and analysed the commit behaviours. Overall they discovered 17 observations.

A general finding is that developers are preferring DVCS over CVCS at a ratio of 65% to 35%. They highlighted SVN and Git for each type of CVS in their study. The first observations target the size of the commit. There are more smaller commits on DVCS and no significant change in migrated systems. It is easier to determine the change-set in Git and the potential of conflicts is lower than in CVCS. This results in fewer "end-of-business"-day commits.

Another observation refers to the splits of commits. There are more splits of commits in DVCS because of the better handling of the selection of changed files in Git. The developers are able to contribute more details of the changes regarding the implementation of new features.

The sixth observation is about the assembly of commits. At the local repository the developers can often commit and squash unnecessary commits together before pushing them to the server. Most developers explained that

⁸<https://github.com> (Accessed on: 19.10.2017)

⁹<https://bitbucket.org> (Accessed on: 19.10.2017)

3 Reference model

they want to have a clean log of the repository and only relevant changes. This is not possible in CVCS.

The next group of observations tries to detect the preferences regarding the type of CVS. DVCS, the newer and more advanced technology, tends to become the favourite CVS for developers. On the other side CVCS has a simple structure and is easy to understand. It was earlier on the market, so most developers are familiar to it. Accustomed procedures are not easy to change to new ones, particularly when they target the behaviour of developers. The observation of commits per day shows an interesting fact: There are no differences in CVCS and DVCS except for the time frames 'per hour' and 'per minute' where commits in CVCS were performed 4-5 times more often. These are the most relevant time frames.

At the end there are observations regarding the team size and Issue Tracking Systems (ITS) like Jira in the projects. All team sizes prefer DVCS systems and manage their change-requests in ITS. It is more common to use issue tracking numbers in commit messages. Commit messages with tracking numbers appear a little more often in DVCS (Brindescu et al., 2014).

If the introduction of a version control system is required, it is highly recommended to use a distributed version control system like Git or Mercurial. This supports the approach of Continuous Delivery with its characteristic commit frequency.

The commit to the code repository is followed by the build of the application.

The build phase requires two parts, the build tool and the build server. The build tool is responsible for the way of building the application and the build server executes it from an independent location, not from a developer's workstation. Unit tests are important for the stability of the application and they wrap up the build phase.

The build tool manages the dependency and the build process. The build process consists of steps like compiling the code and creating the binaries that are deployed later. Build tools are important to ensure the reproducibility. In most cases the unit tests are part of the build process. Unit tests cover the basic functionality of the application by checking for the right variable types and function return values.

Depending on the programming language or used framework there are

3.2 Preparation

multiple alternatives to choose from. As for projects using the programming language JAVA the most common build tools are 'Ant', 'Maven' and 'Gradle'. All of the three have different characteristics but all are capable to fulfill complex builds. The frequently used framework 'Spring' supports Maven and Gradle¹⁰. Ant allows to declare build scripts in a XML-file and to execute the targets through a command shell. Ant instructions are based on the command line set and can be combined. There is no direct support for dependencies.

Maven follows a declarative approach. Instead of writing build scripts there are more basic restrictions and guidelines to meet. Maven has a default lifecycle by convention¹¹, which for example includes the steps compile, install, test and deploy. The lifecycle can be modified with plugins. The Project Object Model file ("pom.xml") contains all definitions of the build. Changes to the configuration can be made there. At the beginning of the file the developer configures the main information like artefacts, version and packaging type. The file also has integrated parts for the dependencies that indicate dependent artefacts with version and scope. Additional plugins can be added at the end.

Gradle is the newest build tool of those three and tries to combine Ant and Maven. It is based on the convention and configuration with added flexibility. Gradle is written in Groovy¹² and specified as a Domain Specific Language (DSL). Groovy relates to JAVA and simplifies it for the usage of Gradle. This allows Gradle to integrate build scripts easier. Gradle determines a dependency tree for the tasks. It only processes a step in case of a change. It is also possible to add Gradle and its version with a wrapper to the sources. That way it is included in the version control and ensures that all developers work with same Gradle version. This feature helps to keep the reproducibility in every context, developers' environment and build server (Wolff, 2014).

Apart from JAVA based projects, there are some that are using other programming languages, frameworks or patterns. An additional common approach is the usage of Node.js¹³ for microservices in order to realize a

¹⁰<https://spring.io/guides/gs/spring-boot> (Accessed on: 23.10.2017)

¹¹<https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html> (Accessed on: 23.10.2017)

¹²<http://groovy-lang.org> (Accessed on: 23.10.2017)

¹³<https://nodejs.org> (Accessed on: 25.10.2017)

3 Reference model

REST-API with a minimized GUI and a database connection. Node.js provides an integrated package manager 'npm' and stores all information for a project in a JSON-file called "package.json". This configuration file has similarities to the "pom.xml"-file of Maven. It manages the dependencies of packages and supports predefined commands for running and testing.

Other programming languages and frameworks like Laravel¹⁴ in PHP work with a similar approach.

Unit tests are necessary to ensure a good quality and robustness. It is highly recommended to perform and add them to the build. The test of the smallest units like methods and functions provides a first test coverage. A good approach to integrate unit tests is Test-Driven-Development (TDD) where the developers write the tests simultaneously to or ahead of the implementation. While the development process is still on-going they are able to test the changed source code with the previously written tests (Louridas, 2005).

The test of the basic functionality is preferably a part of the build process. Further tests of the application that cover more the coherence of the units can be executed in the next phase called "Test". Relations and dependencies between components are simulated through stubs and mocks in the first test phase. A mock object is a local representation of the dependent component and reacts to function calls in the same format as the original object (Fowler, 2007).

When a microservice is written in JAVA there is the option to use JUnit¹⁵. This is open-source and wide-spread in the community. There is also the possibility to use mock frameworks like mockito¹⁶ to easily create mocks for the tests.

Similar to the frameworks in JAVA test frameworks exist for Node.js related projects with Mocha.js¹⁷ and PHPUnit¹⁸ for PHP projects.

Additional types of tests during the test phase extend the basic coverage of the unit tests. More specific tests are for example acceptance or performance

¹⁴<https://laravel.com/> (Accessed on: 25.10.2017)

¹⁵<http://junit.org> (Accessed on: 27.10.2017)

¹⁶<http://site.mockito.org/> (Accessed on: 27.10.2017)

¹⁷<http://mochajs.org> (Accessed on: 28.10.2017)

¹⁸<https://phpunit.de> (Accessed on: 28.10.2017)

tests.

A performance test targets the robustness and gives some indication for the required scalability of the components. The main tasks are to determine the performance and the throughput. Performance describes how fast the application processes a task. The throughput observes how many requests can be handled by the system (Wolff, 2014).

A popular framework for performance tests is JMeter¹⁹. With JMeter the user is able to define a test plan to be executed at a specific time. The performance tests by JMeter can be integrated in the build tool or as post-build action at the build server²⁰. Other performance test frameworks are Gatling²¹ or Grinder²².

Acceptance tests are more complex than performance tests, because they try to cover and to ensure the requirements of the customer in an automatic way. A well-known framework for acceptance tests is Selenium²³ which is available in two versions. It is possible to define a test by an IDE as a browser plugin or through an API. Tests defined by the API are recommended because they can be included in the pipeline and are executed without a browser. Selenium works like a remote control for a browser. Form controls of a website are identified by the *id*-Tag and are assigned to pre-configured inputs. After the submission of the form the *body*-Tag is searched for a success-message. This is a simple example of the usage of an acceptance test in selenium.

Other test frameworks being suitable for acceptance tests, are Cucumber²⁴ for PHP or Mocha²⁵ for Node.js.

In the end it might be helpful to add manual tests by testers to cover the specific use-cases and retrieve feedback about the handling and usability of the application.

¹⁹<http://jmeter.apache.org> (Accessed on: 29.10.2017)

²⁰<https://www.blazemeter.com/blog/continuous-integration-101-how-run-jmeter-jenkins>

²¹<https://gatling.io> (Accessed on: 29.10.2017)

²²<http://grinder.sourceforge.net> (Accessed on: 29.10.2017)

²³<http://www.seleniumhq.org/>

²⁴<https://cucumber.io> (Accessed on: 29.10.2017)

²⁵<http://mochajs.org> (Accessed on: 28.10.2017)

3 Reference model

The last task targets the documentation of the services. In this step the documentation must be checked if all information has been recorded and updated to the latest decisions.

3.2.2 Preparation of the Infrastructure

The preparation of the infrastructure includes the selection of a remote or local code repository and the installation of a build server. A detailed view of the process can be seen in figure 3.11.

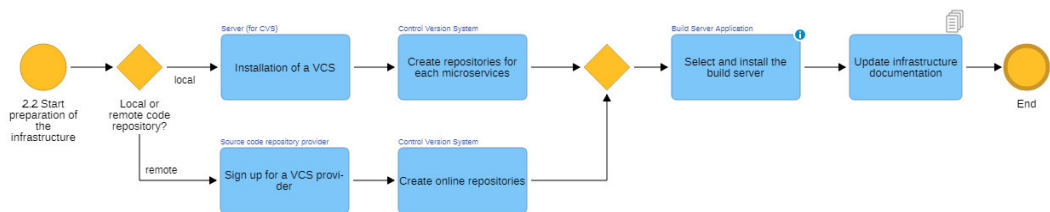


Figure 3.11: Preparation of the infrastructure

After the decision for the version control system in section 3.2.1 it is necessary to determine if the code repository should be kept on local servers or bought from providers online.

This decision can depend on the budget, the provided resources of the project and the preference to hold the data in-house. If the decision is made to create a local repository, a version control system must be installed on a dedicated server in the company. This allows full control over the handling of the repositories.

To sign up for online repositories and to pay a fee is another option. The realization takes by far less time and apart from the fee there are no costs for server hardware nor for maintenance.

At the end the developers must be provided with remote repositories so they can synchronize their local repository with the remote one (Wolff, 2014).

The build server or Continuous Integration (CI) server is one of the most important parts of the pipeline. It ensures the reproducibility by being

3.2 Preparation

separated from the developers' environments and only relies on checked-in sources from the code repository and the server infrastructure. No manual steps are allowed to the developers. Server environment specific adaptations must be defined in build scripts. Builds can be triggered at a certain time or whenever a commit is made.

Then the sources are built and tested. Afterwards the CI server is able to process the next steps of additional tests or deployment (Humble and Farley, 2010).

There are several available solutions for the introduction of a build server. The most popular solution is the open-source application *jenkins*²⁶ which is under MIT license. It offers many free tutorials and supports a large amount of plugins. Jenkins is written in JAVA and can be run in a java virtual machine like Tomcat. It provides several steps for build and deployment from the core functionality. Build jobs can be created and managed by a graphical user interface in a browser. There is a number of further open-source applications²⁷ available.

Another solution for example is *Bamboo*²⁸ which has been developed by Atlassian and uses the same technology as *jenkins*. It is extensible through add-ons and requires a license for commercial purposes.

Visual Studio Team Services²⁹ by Microsoft is also a solution offering build and deployment mechanism. Its documentation provides multiple tutorials³⁰ for the deployment of different scenarios.

On the market a lot of solutions have been evaluated regarding their strength and market position³¹. Depending on the previous decision for the architecture and the programming language a build server can be chosen.

In the end the documentation, which was introduced in section 3.1.2, should be updated with the latest decisions.

²⁶<https://jenkins.io> (Accessed on: 30.10.2017)

²⁷<https://www.blazemeter.com/blog/jenkins-vs-other-open-source-continuous-integration-servers> (Accessed on: 30.10.2017)

²⁸<https://de.atlassian.com/software/bamboo> (Accessed on: 30.10.2017)

²⁹<https://www.visualstudio.com/de/team-services> (Accessed on: 30.10.2017)

³⁰<https://docs.microsoft.com/de-at/vsts/build-release> (Accessed on: 30.10.2017)

³¹<https://www.g2crowd.com/categories/continuous-integration> (Accessed on: 30.10.2017)

3 Reference model

3.3 Configuration

The configuration is the last phase of the reference model. The two layers, software (3.3.1) & infrastructure (3.3.2), are working together and target the creation of the build jobs at the continuous integration server. It must be defined how and to which environment of the infrastructure the builds should be deployed.

3.3.1 Configuration of the software architecture

The steps of the configuration on the software side are shown in figure 3.12.

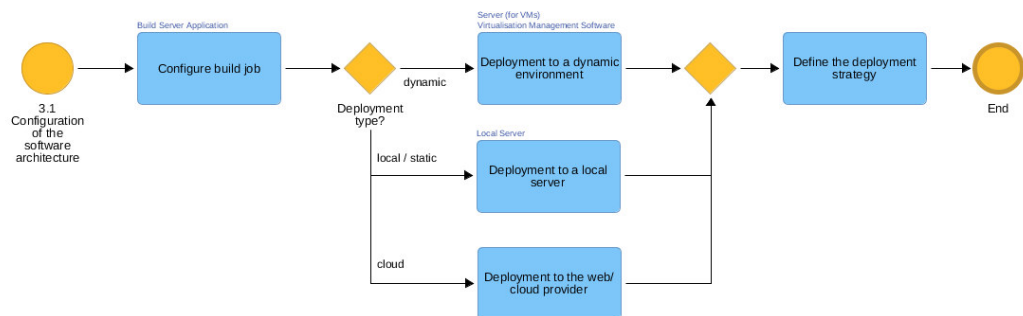


Figure 3.12: Configuration of the software

A build job must be defined for each microservice. It could also be helpful to create a build job that rebuilds all involved microservices of the system and therefore supports a completely clean installation.

In general the configuration of a build job is divided into three parts:

- Pulling the code from the source code repository
- Process of the build
- Configuration of post-build-actions

3.3 Configuration

At first the build job needs access to the version control repository to be able to pull the changes of the source code. In most cases it is possible to trigger the job for specific time frames or even to add a hook to the commits of the version control system. This allows the run of a build for every push to the server.

After the source code has been pulled from the repository the build server creates the binaries by using the selected build tool. The common build tools are supported by most build servers.

What targets or commands have to be executed in the build tool has to be configured in the build job. For maven for example there is the target *mvn package*. The commands can also be chained together to add more steps like tests.

As soon as the binaries are available in the workspace of the build server they are ready for the deployment.

The post-build-actions allow the integration of further tests such as performance and acceptance tests. Finally a deployment alternative can be added ((Fowler and Foemmel, 2006) & (Wolff, 2014)).

There are different deployment types to choose from. They are visible in the process 3.12 as the second task of this phase.

As for the implementation process of an application each developer usually has his own development environment with an editor, workspace and further configurations. One thing all members of the team have in common is the access to the source code repository. Commits should result in a new build at an independent server. Most of the time this is a test server.

The continuous integration server pulls the source code from the repository, builds the binaries if necessary, executes defined tests and afterwards deploys the application to the destination. Depending on the chosen technology a build is required. For some approaches the source code needs only to be copied to the target folder.

The deployment is categorized in three types: local server, dynamic infrastructure and cloud / web-hosting.

With the help of post-build-actions the resulting binaries can be processed further. After the test phase is completed the deployment can be proceeded. Through plugins of the build server the binary (for example a *.WAR-file)

3 Reference model

can be deployed directly to a tomcat instance in the same network. Another option is to copy the results of the pipeline to a target folder of a local server in the network.

A newer and more efficient way is to supply the infrastructure with a dynamic approach by creating a virtual machine and deploying to it. With the software Vagrant³² the user is able to manage the virtual machines. This makes the addition and removal of virtual machines easier. The newly created virtual machine has to be configured with build scripts then. Chef³³ allows to install and to set up the system on the virtual machine by defining a configuration script. Figure 3.13 shows the structure of the integration of a dynamic infrastructure (Wolff, 2014).

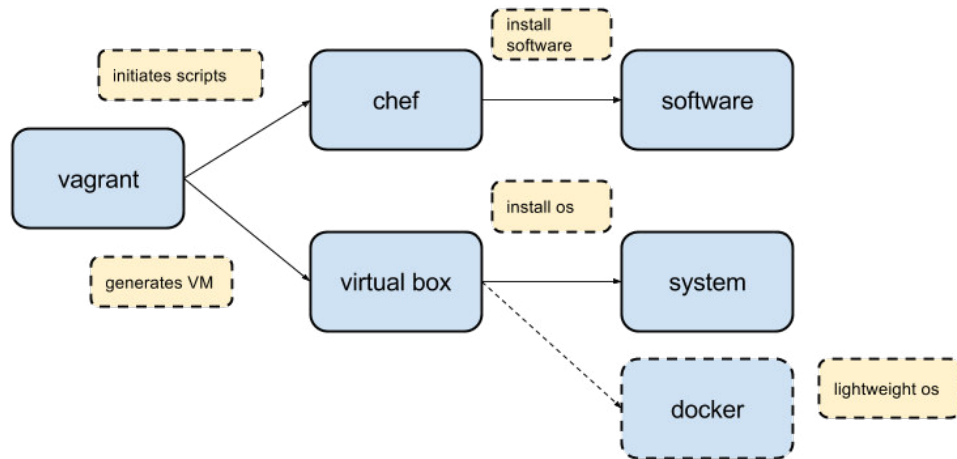


Figure 3.13: Dynamic infrastructure with Vagrant, Chef and Docker. Based on Wolff (2014)

The created virtual machines have a complete operating system installed. The software Docker³⁴ tries to minimize this by providing containers including the necessary features to run the application. They are Linux-based and can be considered as a light weighted operating system where containers share a mutual kernel.

³²<https://www.vagrantup.com/> (Accessed on: 05.11.2017)

³³<https://www.chef.io>

³⁴<https://www.docker.com>

3.3 Configuration

Apart from a deployment to a local server or to a dynamic infrastructure it is also possible to integrate the release to a webhosting or a cloud environment through additional plugins of the build server. Popular cloud hosting providers are Amazon Web Services³⁵ or Heroku³⁶ supporting different types of deployment like the deployment to a virtual server or the creation of a container in the cloud. They provide the deployment for several programming language based applications. Most of them supply the customers with documentation³⁷ of the setup.

With the help of the FTP-protocol the build server is able to simply upload the binaries to a usual web host environment.

The deployment either requires a downtime or continues even without the loss of access to the application.

Usually an application is first deployed to a test and later to a production environment. For a new release the test environment will be upgraded to the new version and the user acceptance tests will be performed. Once the users have completed the acceptance tests the upgrade to the production environment can be executed. This cycle repeats for every new release.

The blue-green deployment approach uses two servers and they are labelled as blue respectively as green. After the upgrade of the blue server to the new version and successful tests it switches to productive through a system address change. The green server keeps the old version until the next release date. At that moment the procedure will be repeated for this server. The advantages are that there is no downtime of the application and that it is always possible to switch back to the previous version. A disadvantage is the hardware costs for running two servers, which is similar to the first approach. Changes to the database structure are not easy to handle for this approach.

If the application is spread over multiple servers via load-balancing a canary releasing can be used. Canary releasing targets the upgrade of single units

³⁵<https://aws.amazon.com> (Accessed on: 05.11.2017)

³⁶<https://www.heroku.com> (Accessed on: 05.11.2017)

³⁷<https://aws.amazon.com/de/getting-started/resource-center> (Accessed on: 05.11.2017)

3 Reference model

within the cluster. This results in a partial upgrade of the system allowing only some users to apply the new features. If no problems are occurring during a certain time of observation the other nodes of the cluster will be upgraded as well. This is achieved with the help of logging and monitoring the system. The advantages are that erroneous releases do not bother all users and it is easy to roll back the test nodes (Wolff, 2014).

In a particular use-case it might be possible to integrate a continuous deployment where every change is pushed directly to the productive environment. This requires a large amount of trust between the developers and the end-users what has been previously discussed in section 2.3.3 (Humble and Farley, 2010).

3.3.2 Configuration of the infrastructure

The last step of the reference model, the configuration of the infrastructure, is documented in the process 3.14.

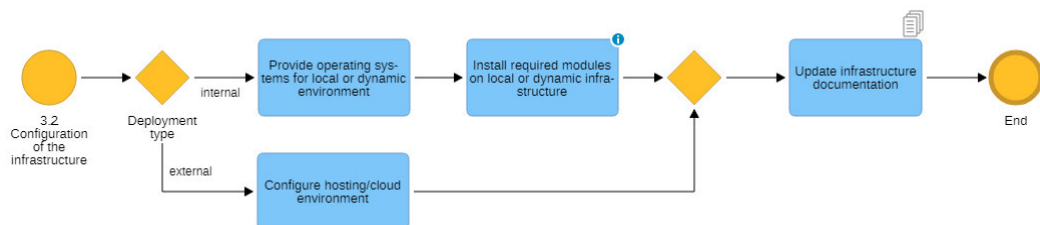


Figure 3.14: Configuration of the infrastructure

This process depends primarily on the decision of the deployment type and the selected programming language of the build tool.

For an external deployment in the cloud or web hosting there might be the need to configure settings for the environment regarding the provided modules, features or permissions.

The upper decision path of the process targets the internal deployment type, which is realised with in-house components and hardware.

3.4 Reference architecture and infrastructure

This relates to the previous section about the configuration of the build jobs. The provision of dynamic infrastructure with build scripts has been mentioned earlier.

The local static servers require an operating system and furthermore specific modules like webserver or webcontainer installation.

If a dynamic infrastructure has been chosen, it is possible to select a pre-defined virtual box with pre-installed components. Additional customized configuration can be achieved with build scripts including shell commands. For example, there is a virtual box template with a unix-based operating system. Through the build script the webserver or webcontainer such as apache or apache tomcat is installed.

This is similar to the approach with containers like docker. The containers need a light weighted template and equally a build script for the components (Wolff, 2014).

A common tool for various types of virtualization is vagrant cooperating with virtual box, chef and docker. Figure 3.13 shows how the tools work together. The software architecture and the infrastructure work closer together in this phase. Once the build jobs of section 3.3.1 are configured and the decision about the deployment in this section is made the documentation needs to be updated in its specific file.

3.4 Reference architecture and infrastructure

When all three phases of the reference model are completed the decisions of the process models result in two reference models for each layer. The abstracted form of the software architecture layer is shown in figure 3.15 and the resulting infrastructure is presented in figure 3.16. Together with the documentation of the microservices and the infrastructure the reference models provide a good overview of the resulting environment.

The reference software architecture consists of the deployed server and the web-containers holding the deployed binaries. These exchange information through the communication pattern like REST. Besides the usage of multiple deployed servers the cardinalities show how web-containers

3 Reference model

and the contained microservices divide the deployment. A server can have multiple web-containers where one or more binaries are located. Multiple web-containers offer the possibility to start or to shut down only one specific web-container. Using a single web-container has the advantage of reduced management efforts.

Each microservice provides a communication interface. It also requires a build tool to manage the dependency and the test of the selected framework.

The infrastructure is separated into three components: build server application, version control system and provision of the deployment. The build server application needs an independent server and in most cases a web-container at run-time. The control version system holds the repositories. It is possible to sign up at a provider or to keep the data at a local installation. The deployment has various options. For a dynamic approach a virtualization solution is required which manages the virtual machines and the containers. The second option is the deployment to a web or cloud provider. The last option is the set-up of a local environment by providing a static server.

3.4 Reference architecture and infrastructure

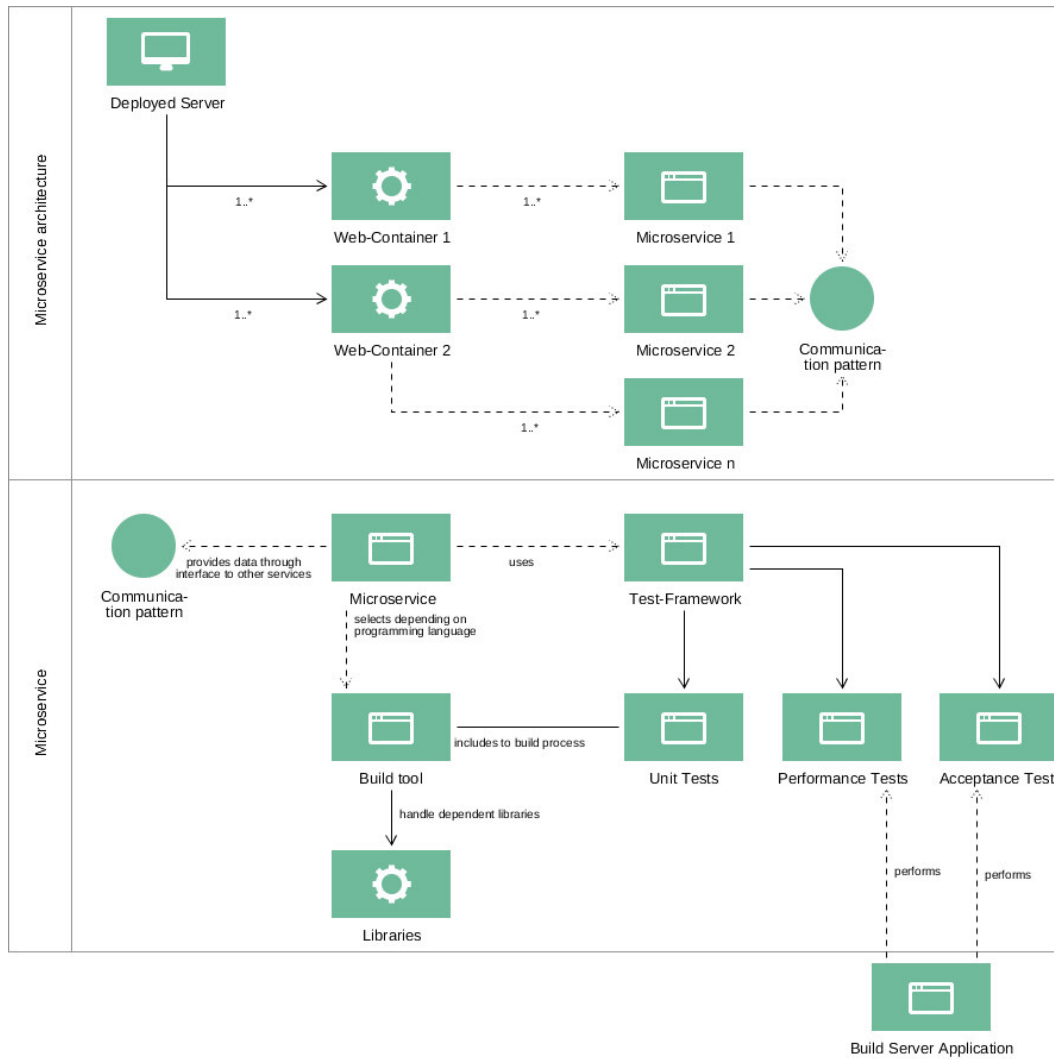


Figure 3.15: Reference software architecture

3 Reference model



Figure 3.16: Reference infrastructure

4 Evaluation

The goal of this chapter is the adaptation of the previously described reference model to a realistic example and the evaluation of its results. The website of the Open Models Laboratory at the University of Vienna has been chosen for doing the evaluation.

Section 4.1 presents the Open Models Laboratory and section 4.2 shows how the reference model is applied to this website. Section 4.3 provides a summary of the evaluation.

4.1 Open Models Laboratory

The Open Models Laboratory (OMiLab)¹ is located at the University of Vienna in the institute of informatics. It is run by the department of knowledge engineering².

The OMiLab operates in the research field of modelling methods. Thus, it provides the possibility to create own modelling methods and it supports the whole creation process with guidance.

A modelling method has two major parts, the mechanisms and the modelling technique. The modelling language and the modelling procedure sum up to the modelling technique. Figure 4.1 explains and visualizes these terms and their relation to each other (Karagiannis and Kühn, 2002).

OMiLab focuses on the creation of the modelling language and supplies

¹<http://www.omilab.org> (Accessed on: 19.05.2018)

²<http://ke.cs.univie.ac.at> (Accessed on: 19.05.2018)

4 Evaluation

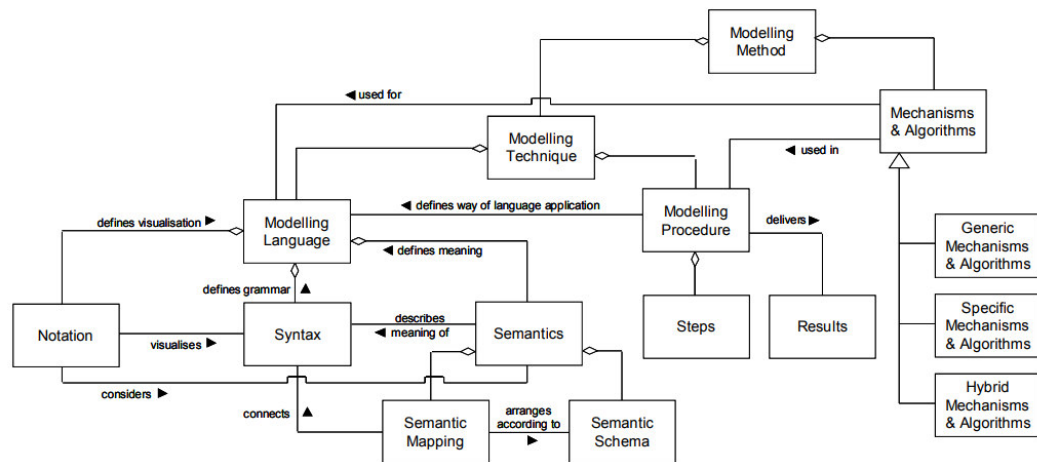


Figure 4.1: Components of a metamodeling method (Karagiannis and Kühn, 2002, p. 3)

mechanisms in their meta modelling platform tool ADOxx³. The combination of both leads to a final modelling method. The main objective of the users is the realization of a modelling tool for the problem domain they operate in. They create a concept to find a representation of the domain. The main task is the definition of a modelling language describing a model that can be instantiated later in ADOxx (Fill and Karagiannis, 2013). Figure 4.2 shows the metamodeling language levels.

It starts from the bottom with the original. This can be any object in the real world. The model is an abstraction of the original. The model is based on a metamodel and built with a modelling language (Karagiannis and Kühn, 2002). The OMiLab provides the ADOxx tool to the users to create a meta-model and as a result to have a modelling tool with their own modelling language (Fill and Karagiannis, 2013).

One example for the abstraction of the model levels is a subway network. The real network itself is the original. A model of this subway network is the subway map. The metamodel in turn consists only of the stations and connecting lines and provides these definitions to the modelling language for modelling a subway map in the end.

As the metamodel is very flexible in its definition it can fit several different

³<https://www.adoxx.org> (Accessed on: 28.07.2018)

4.2 OMiLab: Evaluation of the reference model

problem domains and provide various modelling languages.

Common modelling languages are Event-Process-Chain (EPC), Business Process Modelling Notation (BPMN), Unified Modelling Language (UML), Petri-Nets and Entity-Relationship-Model (ERM).

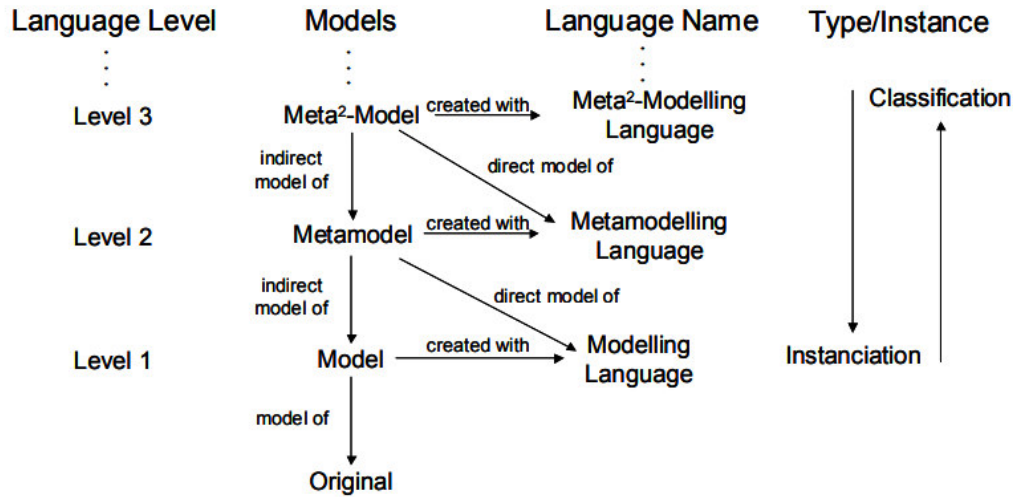


Figure 4.2: Metamodelling based on language level (Karagiannis and Kühn, 2002, p. 3)

The OMiLab is an open community for all people world-wide who are interested in this research field. By offering also information about upcoming events, publications and most importantly the submitted final modelling methods of the contributors the website represents a main contact point, too.

The web application of the site is organized in a microservice architecture with a basic approach of the Continuous Delivery. This is the reason to use it to evaluate the reference model of the previous chapter 3 (Götzinger, Miron, and Staffel, 2016).

4.2 OMiLab: Evaluation of the reference model

This section evaluates the reference model by using the web-application of the OMiLab. It is divided into the three phases of the reference model:

4 Evaluation

planning, preparation and configuration. It is also separated in the main layers software architecture and infrastructure.

4.2.1 Planning

The planning phase of the reference model's software architecture covers the split of the microservices, the communication pattern and the definition of the microservices. The view of this process is available in section 3.1.1 whereas 3.1.2 shows the process of the infrastructure deals with the provision of the deployment systems, the build server and the database server.

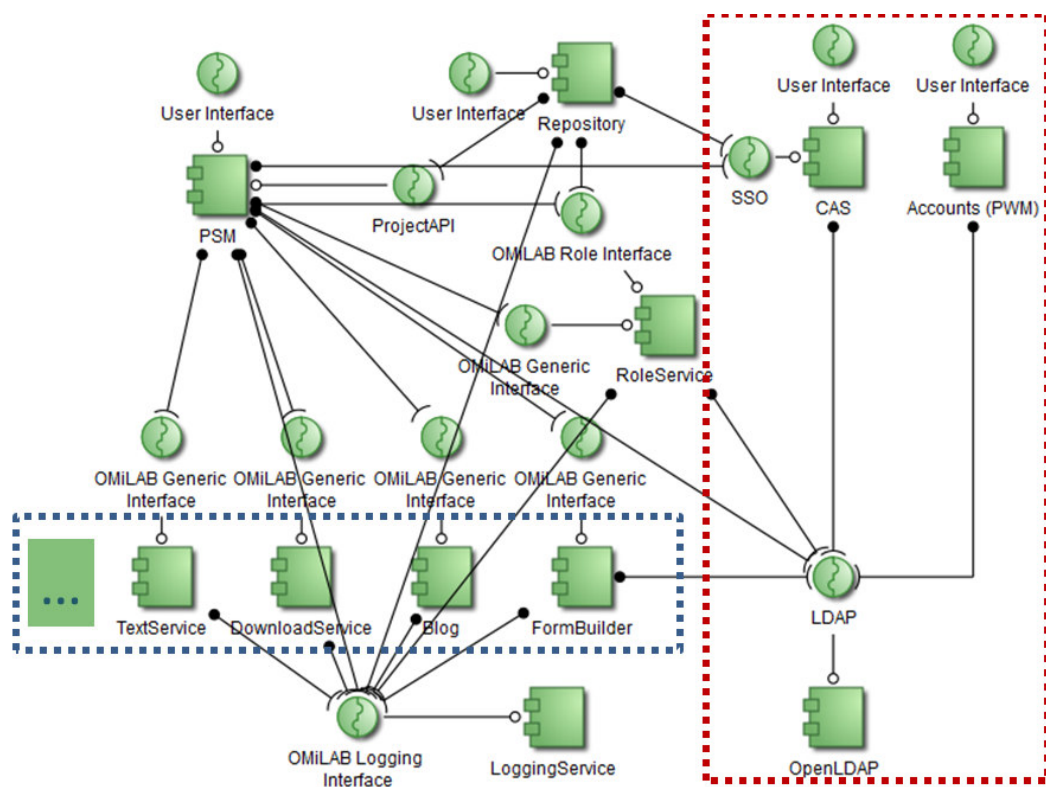


Figure 4.3: OMiLab architecture (Based on Göttinger, 2015, p. 29)

4.2 OMiLab: Evaluation of the reference model

Figure 4.3 by Götzing (2015, p. 29) visualizes the architecture of the software in the OMiLab and therefore the result of the split of the microservices. The Project Structure Manager (PSM) is the major service having connections to all other services. Other services of high importance are the Repository, the RoleService and the LoggingService. The Repository provides the file management in the system. The RoleService manages the roles and responsibilities. The LoggingService relates to all services of a central log system. The services of the red framed area handle the user and the account management. The services in the blue box provide functionality for the pages of the web-application such as text, forms and downloads. This part of the architecture is designed to be flexible and extensible for additional use cases of web-pages.

In summary, the split approach starts with a main service (PSM), and all other services for files, roles and logging are added immediately to provide the basic functionality.

In addition there are smaller representational and interchangeable services for the content of the web-site. These services can be assigned to the projects and help to represent content in several ways. Some services are loosely coupled and some have stricter dependencies to each other. It is difficult to make them fully independent. The deployment of a service can be performed without the others because each service is deployed in its own WAR-file.

The selected communication pattern is the Representational State Transfer (REST) interface which allows the services to cooperate. Each service provides functionality through its endpoints.

All existing microservices are developed in JAVA with the framework Spring Boot and additional plugins though there are no restrictions for the programming language. Also, a different technology stack can be used for the backend.

A Graphical User Interface (GUI) is present and implemented via Thymeleaf⁴ and Bootstrap⁵. Thymeleaf is a template engine supplying the data from the back-end to the front-end. Bootstrap is a front-end framework with

⁴<https://www.thymeleaf.org> (Accessed on 21.05.2018)

⁵<https://getbootstrap.com> (Accessed on 21.05.2018)

4 Evaluation

client-side scripting and stylesheets for creating a user interface.

Currently all microservices use the same technology stack. This allows the re-use of tools and the required knowledge being available for all services. It is recommended to use the same technology stack so the services benefit most from the synergies.

The last step of the planning of the software architecture is the documentation. To complete a simple form is sufficient. Figure 4.4 shows how a documentation template could look like.

The infrastructure is the second process of the planning phase. The application needs first the infrastructure for the deployment. The University of Vienna provides two virtual servers, one for a productive and one for a test environment, from the local university cluster. Both systems are static and manually configured.

For the next task an additional third server is available to be used for the build server. As most build server applications need a web-container for the deployment the server has almost the same configuration as the first two. The final decision for the build application is handled in the preparation section 4.2.2. The third task is the requirement for a database system. Each server will be configured later with a mysql database instance for each one. An example of a documentation of the OMiLab infrastructure could look like figure 4.5.

The organization is not affected as much as in a usual company. The development is realized by the members of the department of knowledge engineering. It is a small team of developers and responsables for the operation who are working closely together. All services are assigned to the same team. The approach of the realization of microservices for the web-site was a well-defined decision. As a result the developers are aware of the new processes of the delivery and of the work behaviour. The development is open-source. Since the OMiLab is located at the University of Vienna, in some courses the students are motivated to contribute to the development of new microservices. The services of the blue box in figure 4.3 can be extended with additional services providing new functionality.

The students brainstorm about a new idea and afterwards group together for the development of a new microservice. Aside from the development they are responsible for the support and the documentation. This makes the

4.2 OMiLab: Evaluation of the reference model

07.08.2018

Documentation



Microservice Documentation PSM

General

Name:	PSM
Description:	Project structure manager
Responsible:	David Götzinger
Responsible (Deputy):	David Götzinger

Technologies

Programming language:	JAVA
Frameworks:	Spring Boot
Build tool:	maven

Interfaces

Interface	Description
REST	Provides functionality through REST endpoints

Tests

Framework:	-
Description:	Basic coverage through the plugin "findbugs"

Deployment

Destination:	Build job to the test system and manual deployment to productive
Description:	The test environment is equipped continuous delivery.

Figure 4.4: Example of a documentation for a microservice

structure of the OMiLab architecture very flexible for new features.

4 Evaluation

07.08.2018

Documentation



Infrastructure Documentation

Version Control System

Type / Technology:	GitLab / Git
Location:	<Server 3>
Responsible:	Michael Bültmann
Description:	Distributed approach chosen with Git and the suite GitLab for additional documentation and issue tracking system.

Build server

Type:	jenkins
Location:	<Server 3>
Responsible:	Michael Bültmann
Requirements:	Apache Tomcat
Description:	The selected build server is jenkins, which requires a web-container for the running. The location is prepared with an apache tomcat.

Deployment

Type:	Test & Productive (+ Dynamic)
Locations:	<Server1>, <Server 2> (+ Virtual machine)
Responsible:	Michael Bültmann
Description:	There are two servers for the test and productive environment. There is also the possibility to generate a virtualbox with the current system.

Figure 4.5: Example documentation of the infrastructure

4.2.2 Preparation

The preparation of the software architecture consists of the decision for the version control system, the build tool and the test framework.

The decision for the version control was made in favour of the distributed

4.2 OMiLab: Evaluation of the reference model

approach with the technology Git. The software tool itself is GitLab⁶. Apart from the version control repository it includes documentation and issue tracking functionalities. The newest version also supports integrated deployment and monitoring. In the case of the OMiLab this is still achieved with Jenkins.

The decision for the build tool is influenced by the selection of the programming language and has to be made for each service. The selected programming language is JAVA and therefore the common framework Spring Boot⁷ was chosen. Spring Boot supports the build tools maven and gradle. The OMiLab project uses maven as the build tool. Maven uses a declarative approach by defining meta data, dependency and plugins in a XML configuration file. Most services include additional plugins and profile management for the configuration file. An example of the "POM.xml" configuration file is seen in figure 4.6.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4  <modelVersion>4.0.0</modelVersion>
5  <parent>
6  <groupId>org.springframework.boot</groupId>
7  <artifactId>spring-boot-starter-parent</artifactId>
8  <version>1.2.5.RELEASE</version>
9  </parent>
10 <groupId>org.omilab</groupId>
11 <version>0.6.0-RELEASE</version>
12 <artifactId>psm</artifactId>
13 <packaging>war</packaging>
14 <name>OMiLAB Project and Service Manager (PSM)</name>
15 <description>Service Aggregator for OMiLAB</description>
16 <url>http://www.omilab.org</url>
17 <organization>
18 <name>Research Group Knowledge Engineering</name>
19 <url>http://www.dke.univie.ac.at</url>
20 </organization>
21 <scm>
22 <connection>scm:git:git://git@gitlab.dke.univie.ac.at:omilab-core-infrastructure/PSM.git</connection>
23 </scm>
24 <properties>
31 <dependencies>
119
120 <profiles>
688
689
690 <build>
725 </project>
```

Figure 4.6: POM.xml of the PSM service ⁸

By selecting Spring Boot and maven the addition of further test frameworks

⁶<https://about.gitlab.com> (Accessed on 22.05.2018)

⁷<http://projects.spring.io/spring-framework> (Accessed on 22.05.2018)

4 Evaluation

as plugins to the maven configuration is possible. For example a simple test or code validation plugin is [findbugs⁹](#), which is used for most services. The integration of the unit test can also be handled through the maven configuration. With further growth of the system it is possible to add more kinds of tests.

Regarding the infrastructure of the location of the code repository it is possible to sign up at hosting providers, but for the OMiLab the decision was made for a local installation. This was the reason for providing the third server where the repositories for each service, the necessary functionality for the documentation and the issue tracking are provided.

The choice for the build server application fell for the open-source software [jenkins¹⁰](#) to be installed on the third server. On a UNIX-based operating system jenkins can be installed through a package containing an integrated service.

Now the version control system with the code repository and the build server application are on the same server. Later the build server needs to access the code repository for creating the build. As these applications are not constantly at high performance, there is no objection to have them on one server.

4.2.3 Configuration

The configuration is the last step to fulfil the deployment process. It targets the build jobs, the deployment destination and the deployment strategy. In this phase the processes of the software architecture and the infrastructure work closely together. The process of the infrastructure [3.14](#) handles the software installation and the component configuration of the target machine. The task of the software architecture described in the flow [3.12](#) is the configuration of the build job with its type and strategy.

The two systems, test and productive environment, have the following components installed: [ubuntu¹¹](#) as operating system, the [JAVA¹²](#) development

⁹<http://findbugs.sourceforge.net/manual/eclipse.html> (Accessed on 22.05.2018)

¹⁰<https://jenkins.io> (Accessed on 22.05.2018)

¹¹<https://www.ubuntu.com> (Accessed on 22.05.2018)

¹²<https://www.java.com> (Accessed on 22.05.2018)

4.2 OMiLab: Evaluation of the reference model

toolkit, a tomcat¹³ web-container and a mysql¹⁴ database.

The installation is done manually. Each system has a tomcat web-container where all microservices are deployed into. Figure 4.7 provides a simple overview of the used systems and their components in the OMiLab environment.

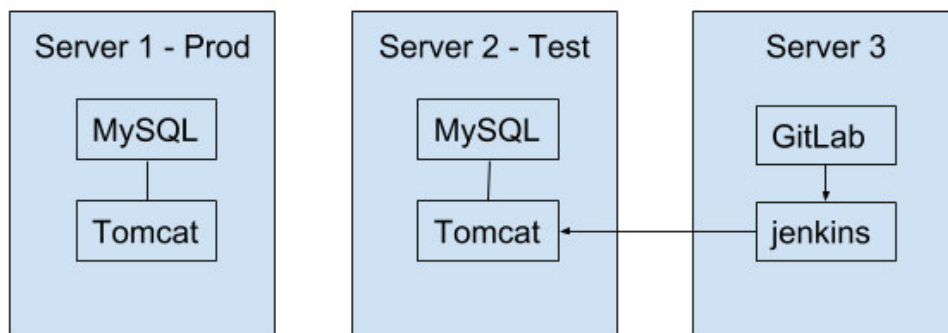


Figure 4.7: OMiLab deployment overview (Based on OMiLab documentation¹⁵)

As part of the evaluation a Vagrant script was created to provide a dynamic environment. The configuration of the virtual machine can be seen in the listing 4.1.

```
1 Vagrant.configure("2") do |config|
2
3   # Box Settings
4   config.vm.box = "ubuntu/xenial64"
5
6   config.vm.provider "virtualbox" do |vb|
7     vb.memory = 5116
8     vb.cpus = 1
9   end
10
11  # Portforwarding
12  config.vm.network "forwarded_port", guest: 8080, host: 6080
```

¹³<http://tomcat.apache.org> (Accessed on 22.05.2018)

¹⁴<https://www.mysql.com> (Accessed on 22.05.2018)

4 Evaluation

```
13 config.vm.network "forwarded_port", guest: 80, host: 6000
14
15 # Synced Tomcat folders
16 config.vm.synced_folder "configs/", "/home/vagrant/configs"
17 config.vm.synced_folder "logs/tomcat/", "/var/lib/tomcat8/logs"
18
19 # Synced Tomcat folder
20 #config.vm.synced_folder "webapps/", "/var/lib/tomcat8/webapps"
21
22 # Installs Tomcat, Mysql and copies configuration-files
23 config.vm.provision "shell", path: "Commands.sh"
24
25 # Creates databases, users and grants permission
26 config.vm.provision "shell", path: "Database.sh"
27
28 # Check the repositories from GitLab, build with maven and copy
  to the tomcat
29 config.vm.provision "shell", path: "Build.sh"
30 end
```

Listing 4.1: Vagrantfile: Provision of the virtual machine

This is a simple provision of a virtual machine, therefore it does not use more powerful applications like puppet or chef that are presented in figure 3.13. The first configuration of the dynamic infrastructure is the basic box selected from the vagrant cloud boxes¹⁶. Here it is "ubuntu/xenial64" containing an ubuntu 16.04 LTS OS similar to provided servers for the OMiLab environment. The next configuration targets the provider which is Virtualbox¹⁷. This is necessary to access the virtual machine which will be listed in Virtualbox once created. Afterwards the forwarded ports between the host and the box are configured, as well as the synced folders between both. The bottom of the listing consists of the provision files which will be executed after the box has been set up.

The *Commands.sh* in listing 4.2 serves for the installation and the configuration of additional components in the newly created box. After the update of the operating system the following main components will be installed: mysql database, java/jdk, tomcat and maven/git. The tomcat receives two

¹⁶<https://app.vagrantup.com/boxes/search> (Accessed on 04.08.2018)

¹⁷<https://virtualbox.org> (Accessed on 04.08.2018)

4.2 OMiLab: Evaluation of the reference model

adapted configuration files which will increase the memory and allow the deployment through the administration management interface.

```
1 echo "##### Configure / Update OS #####"
2 # Update Packages
3 apt-get update
4 # Upgrade Packages
5 apt-get upgrade
6
7 # Install Git
8 apt-get install -y git
9
10 echo "##### Install MYSQL #####"
11 # Set MySQL Pass
12 debconf-set-selections <<< 'mysql-server mysql-server/
    root-password password root'
13 debconf-set-selections <<< 'mysql-server mysql-server/
    root-password.alternatives password root'
14
15 # Install MySQL
16 apt-get install -y mysql-server
17
18 echo "##### Install JDK #####"
19 # Install JDK
20 sudo apt-get install -y default-jdk
21
22 echo "##### Install Tomcat (via package manager)
    #####"
23 sudo apt-get install -y tomcat8 tomcat8-admin
24
25 # Required folder for the OMiLab environment
26 cd /var/lib/tomcat8/
27 sudo mkdir indexes
28 sudo chown -R tomcat8:tomcat8 indexes/
29
30 echo "##### Copy tomcat configuration #####"
31 # Copy tomcat-users configuration for the usage of tomcat manager
32 sudo cp /home/vagrant/configs/tomcat-users.xml /etc/tomcat8/tomcat
    -users.xml
33 # Increase the memory for the tomcat
34 sudo cp /home/vagrant/configs/tomcat8 /etc/default/tomcat8
35
36 echo "##### Restart tomcat8 #####"
```

4 Evaluation

```
37 sudo systemctl restart tomcat8
38
39 echo "##### Install maven #####"
40 sudo apt-get install -y maven
```

Listing 4.2: Commands.sh: Install components and prepare the virtual machine

The provision file *Database.sh* in the listing 4.3 contains the creation of the database for each microservice requiring a database storage. The steps for each database are the import of the schema including the example data, the creation of the specific user and finally the setting of permissions to the user for this database.

```
1 # Create the databases and content for the OMiLab environment
2
3 #psm
4 echo "##### Create Database for Service PSM (db,user,grants)
5 ) #####"
6 mysql -uroot -proot -e "CREATE DATABASE psm;"
7 mysql -uroot -proot -e "USE psm;"
8 mysql -uroot -proot "psm" < "/home/vagrant/configs/sql/psm/schema.
9 sql"
10 mysql -uroot -proot -e "CREATE USER 'psm'@'localhost' IDENTIFIED
11 BY 'password';"
12 mysql -uroot -proot -e "USE psm;"
13 mysql -uroot -proot -e "GRANT ALL PRIVILEGES ON psm.* TO 'psm'@'
14 localhost';"
15
16 #role
17 echo "##### Create Database for Service Role (db,user,
18 grants) #####"
19 mysql -uroot -proot -e "CREATE DATABASE role;"
20 mysql -uroot -proot -e "USE role;"
21 mysql -uroot -proot "role" < "/home/vagrant/configs/sql/role/
22 schema.sql"
23 mysql -uroot -proot -e "CREATE USER 'role'@'localhost' IDENTIFIED
24 BY 'password';"
25 mysql -uroot -proot -e "USE role;"
26 mysql -uroot -proot -e "GRANT ALL PRIVILEGES ON role.* TO 'role'@'
27 localhost';"
28
29 #logging
```

4.2 OMiLab: Evaluation of the reference model

```
22 echo "##### Create Database for Service logging (db,user,
    grants) #####"
23 mysql -uroot -proot -e "CREATE DATABASE logging;"
24 mysql -uroot -proot -e "USE logging;"
25 mysql -uroot -proot "logging" < "/home/vagrant/configs/sql/logging
    /schema.sql"
26 mysql -uroot -proot -e "CREATE USER 'logging'@'localhost '
    IDENTIFIED BY 'password';"
27 mysql -uroot -proot -e "USE logging;"
28 mysql -uroot -proot -e "GRANT ALL PRIVILEGES ON logging.* TO '
    logging'@'localhost';"
29
30 #text
31 echo "##### Create Database for Service Text (db,user,
    grants) #####"
32 mysql -uroot -proot -e "CREATE DATABASE text;"
33 mysql -uroot -proot -e "USE text;"
34 mysql -uroot -proot "text" < "/home/vagrant/configs/sql/text/
    schema.sql"
35 mysql -uroot -proot -e "CREATE USER 'text'@'localhost ' IDENTIFIED
    BY 'password';"
36 mysql -uroot -proot -e "USE text;"
37 mysql -uroot -proot -e "GRANT ALL PRIVILEGES ON text.* TO 'text'@'
    localhost';"
38
39 #download
40 echo "##### Create Database for Service download (db,user,
    grants) #####"
41 mysql -uroot -proot -e "CREATE DATABASE download;"
42 mysql -uroot -proot -e "USE download;"
43 mysql -uroot -proot "download" < "/home/vagrant/configs/sql/
    download/schema.sql"
44 mysql -uroot -proot -e "CREATE USER 'download'@'localhost '
    IDENTIFIED BY 'password';"
45 mysql -uroot -proot -e "USE download;"
46 mysql -uroot -proot -e "GRANT ALL PRIVILEGES ON download.* TO '
    download'@'localhost';"
47
48 echo "##### Flush Privileges #####"
49 mysql -uroot -proot -e "FLUSH PRIVILEGES;"
```

Listing 4.3: Database.sh: Create the database, import the schema and grant permission for users

As the deployment for the OMiLab environment is performed through the

4 Evaluation

build application jenkins, the additional build script in the listing 4.4 provides an alternative way of the deployment and can be executed optionally. It consists of the creation of the workspace and the checkout of the online repositories. Afterwards it is possible to build the binaries and copy them to the tomcat directory. This way allows the setup of the environment in just one step, which is repeatable, as the virtualbox with its content can be created and destroyed immediately. The insight of the deployment process is not as clearly visible as the output of the build jobs in the build applications, therefore it makes senses to use the deployment type for stable releases. There is also the chance to use this approach for special tests, which provision the virtual machine, perform the tests and finally destroy the virtual box.

```
1 echo "##### Create workspace #####"
2 sudo mkdir /home/vagrant/workspace
3 cd /home/vagrant/workspace
4
5 echo "##### Checkout repositories #####"
6 git clone https://gitlab.dke.univie.ac.at/omilab-core-
  infrastructure/PSM.git
7 git clone https://gitlab.dke.univie.ac.at/omilab-core-
  infrastructure/ActivityLoggingService.git
8 git clone https://gitlab.dke.univie.ac.at/omilab-services/
  DownloadService.git
9 git clone https://gitlab.dke.univie.ac.at/omilab-core-
  infrastructure/RoleService.git
10 git clone https://gitlab.dke.univie.ac.at/omilab-services/
  TextService.git
11
12
13 echo "##### Overwrite application.properties #####"
14 # Properties needed different settings, can be solved with an
  additional profile
15 cd /home/vagrant/
16 sudo cp configs/properties/psm/application-distribution.properties
  workspace/PSM/src/main/resources/config/application-
  distribution.properties
17 sudo cp configs/properties/text/application-distribution.
  properties workspace/TextService/src/main/resources/config/
  application-rdfiss.properties
18 sudo cp configs/properties/download/application-distribution.
```

4.2 OMiLab: Evaluation of the reference model

```
19     properties workspace/DownloadService/src/main/resources/config/
    application-distribution.properties
20 sudo cp configs/properties/role/application-distribution.
    properties workspace/RoleService/src/main/resources/config/
    application-distribution.properties
21 sudo cp configs/properties/logging/application-distribution.
    properties workspace/ActivityLoggingService/src/main/resources/
    config/application-distribution.properties
22
23 echo "##### Build binaries #####"
24 ##
25 cd /home/vagrant/workspace/PSM
26 sudo mvn -Pdistribution clean package
27
28 cd /home/vagrant/workspace/ActivityLoggingService
29 sudo mvn -Pdistribution clean package
30
31 ##
32 cd /home/vagrant/workspace/DownloadService
33 sudo mvn -Pdistribution clean package
34
35 ##
36 cd /home/vagrant/workspace/RoleService
37 sudo mvn -Pdistribution clean package
38
39 ##
40 cd /home/vagrant/workspace/TextService
41 sudo mvn -Pdistribution clean package
42
43 echo "##### Stop tomcat8 #####"
44 sudo systemctl stop tomcat8
45
46 echo "##### Copy binaries #####"
47 sudo cp /home/vagrant/workspace/PSM/target/*.war /var/lib/tomcat8/
    webapps/psm.war
48 sudo cp /home/vagrant/workspace/ActivityLoggingService/target/*.
    war /var/lib/tomcat8/webapps/logging.war
49 #sudo cp /home/vagrant/workspace/DownloadService/target/*.war /var
    /lib/tomcat8/webapps/download.war
50 sudo cp /home/vagrant/workspace/RoleService/target/*.war /var/lib/
    tomcat8/webapps/role.war
51 sudo cp /home/vagrant/workspace/TextService/target/*.war /var/lib/
    tomcat8/webapps/textservice.war
```

4 Evaluation

```
52 echo "##### Restart tomcat8 #####"  
53 sudo systemctl start tomcat8
```

Listing 4.4: Build.sh: Create the workspace, checkout the repositories and build the binaries

The summary of the dynamic environment approach is seen in the README.md file 4.8 for the respective GitHub repository¹⁸. This is also featured in the appendix.

¹⁸<https://github.com/michaelb8/omilab-vagrant-deployment> (Accessed on 20.08.2018)

4.2 OMiLab: Evaluation of the reference model

OMiLab Deployment via Vagrant

Provision of a virtual machine with OS, Tomcat and database for the dynamic deployment.

Requirements

- Vagrant: <https://vagrantup.com>
- Virtualbox: <https://virtualbox.org>

VM configuration

- OS: ubuntu 16.04
- Portforwards:
 - 80 to 6000
 - 8080 to 6080
- Logs of the tomcat will be synced in logs\
- configs\ will be synced to the \home\vagrant\configs

Provision Scripts

Shell commands for box installation:

- Commands
 - Updates the OS
 - Installs mysql and jdk
 - Installs tomcat8 and copies tomcat configuration
 - Restarts tomcat service
- Database
 - SQL for each microservice database is stored in configs\sql\ *servicename*
 - Creates database, imports schema, creates user and grants permission
- Build (*optional*)
 - Creates a workspace
 - Checkouts the repository from [DKE GitLab](#)
 - Build the binaries with maven
 - Copy the WAR-files to the tomcat\webapps

Execution

- Browse to the checked-out directory
- *vagrant up* starts up the virtual machine
- *vagrant ssh* allows the SSH Login for further adaptation to the virtual machine
- *vagrant destroy* will delete the virtual machine

Figure 4.8: README.md of the GitHub Repository for the dynamic provision of the OMi-Lab environment

4 Evaluation

The software architecture process targets the build jobs and the deployment strategy. For the time being the build jobs are configured only for the test server. The third server with the version control system (GitLab) and the build application (jenkins) creates the build and deploys it to the second server containing the test environment.

If the manual tests are successful, the deployment to the productive environment will be performed manually as well.

The build jobs are configured at the build server jenkins and the settings consist of three main steps: source code repository, build and post-build actions.

4.2 OMiLab: Evaluation of the reference model

The general configuration in Jenkins is shown in figure 4.9. A description can be added and the configuration for the job can be set.

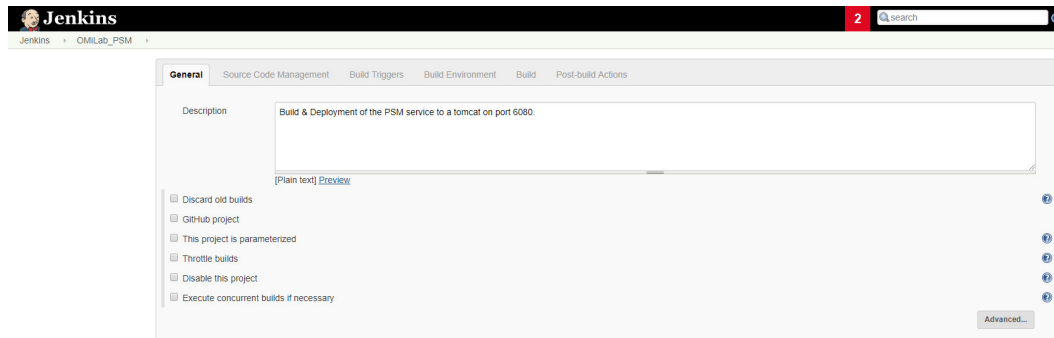


Figure 4.9: Jenkins: Configure job - General

The next figure 4.10 shows the source code management configuration. Multiple version control systems are supported. As GitLab is used, the version control system is Git. For the repository the URL, the branch and for private repository the credentials are required.

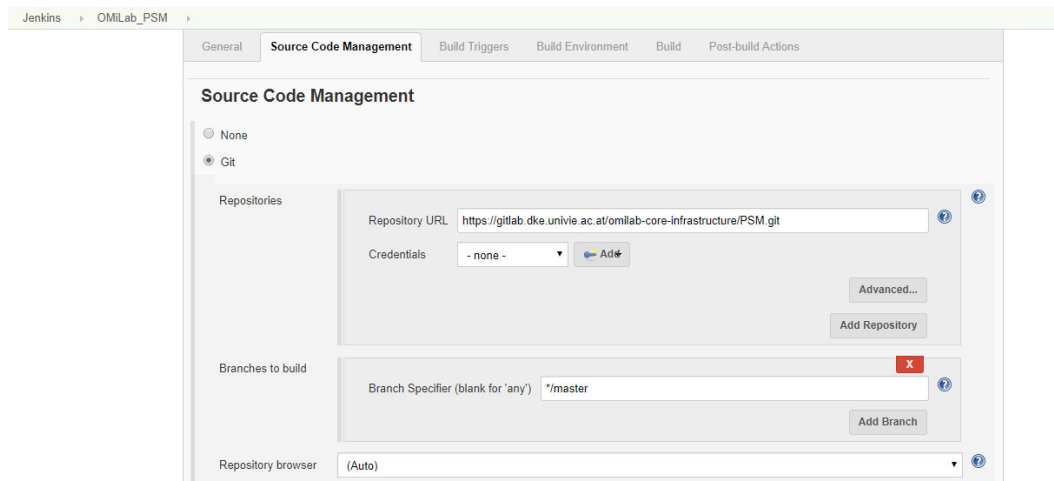


Figure 4.10: Jenkins: Configure job - Source Code Management

4 Evaluation

The second step of the build job configuration is the build and starts with the build trigger. Figure 4.11 displays the options for the build trigger. A hook can be added to the Git repository to trigger a build. As shown in the figure, it is also possible to set a period (every 15 minutes) when the build server checks for changes in the source code management. The build environment allows configuration concerning the build process itself.

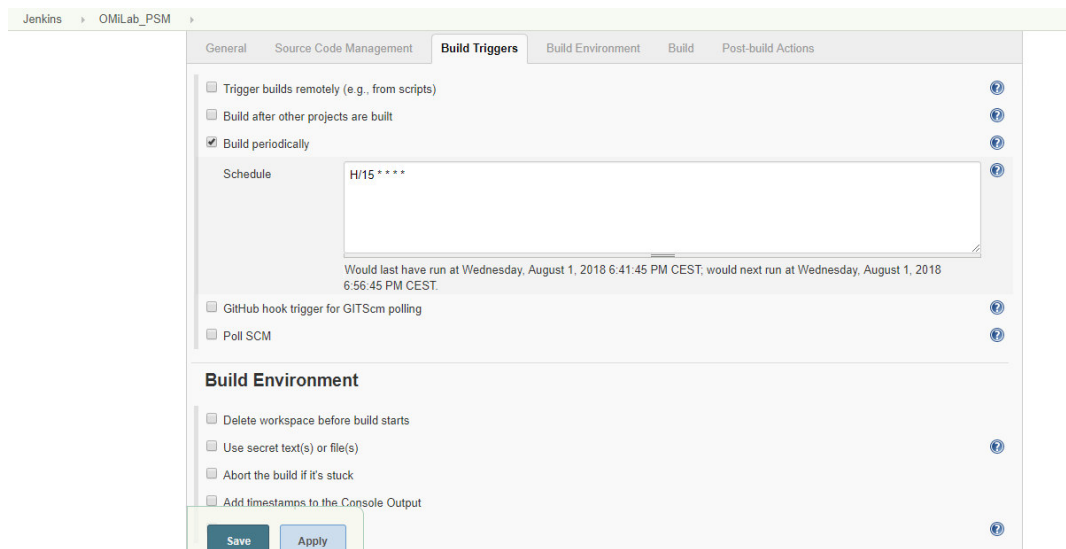


Figure 4.11: Jenkins: Configure job - Build Triggers and Environment

The main step of the configuration of the build job is the build. Figure 4.12 shows the tasks of the build being performed by the build application jenkins. For the deployment to the virtual machine the properties file of the maven project needs some changes concerning the paths. Otherwise the first task of the build would not be necessary. The following build command expects a maven command for its life-cycle, for example *package*. Until this step the sources are pulled to the local workspace of the build job and the binaries are built depending on the maven command. Here the workspace will be cleaned first and the package will be created with the profile *distribution*. After the creation of the binaries, the WAR-file is renamed, because it contains the version number and profile. This is

4.2 OMiLab: Evaluation of the reference model

necessary to keep the URL consistent.

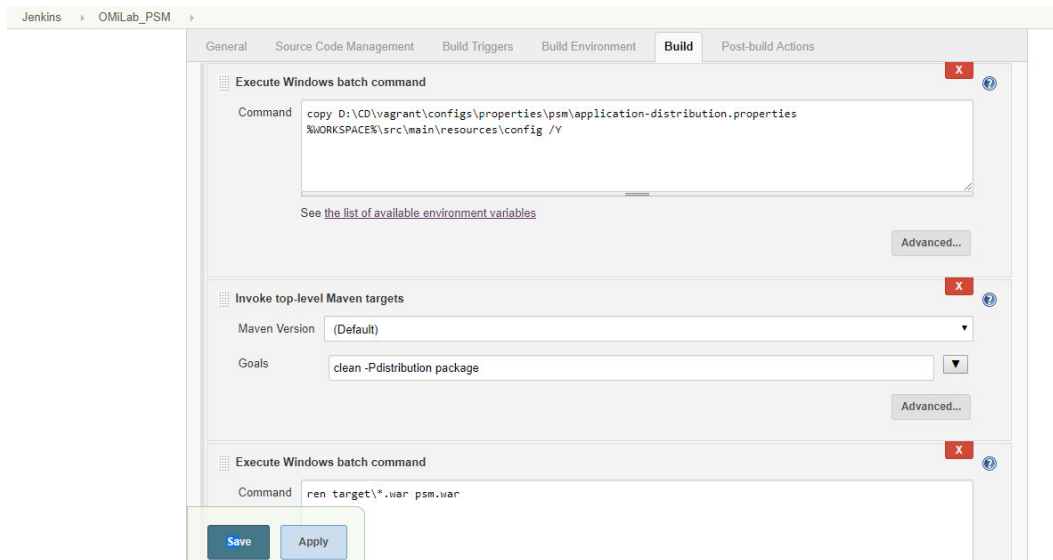


Figure 4.12: Jenkins: Configure job - Build

Afterwards the post-build actions can be defined as shown in figure 4.13. With the help of the plugin *"deploy war/ear to a container"* the built WAR-file is grabbed and deployed to the URL of the specified tomcat. For this step the credential for the manager tool of the tomcat is necessary, which is defined in the *tomcat-users.xml*. There are also options to send e-mail notifications or trigger the next build job.

4 Evaluation

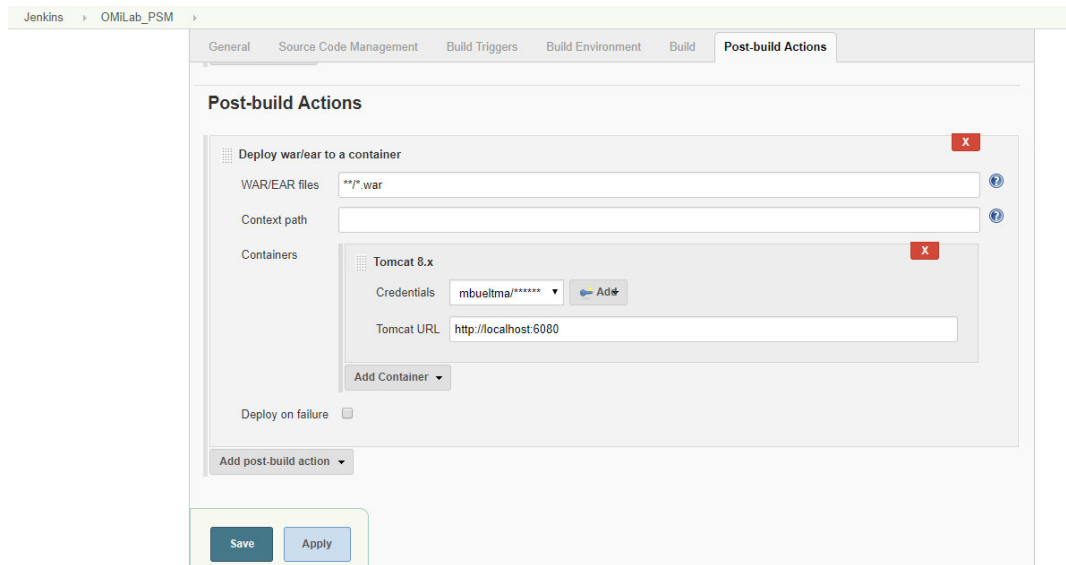


Figure 4.13: Jenkins: Configure job - Post-build

For chained build jobs the plugin *build pipeline* presents a pipeline in a well structured view (figure 4.14). The history of the last runs as well as the execution time and the state of each build job are shown.

4.2 OMiLab: Evaluation of the reference model

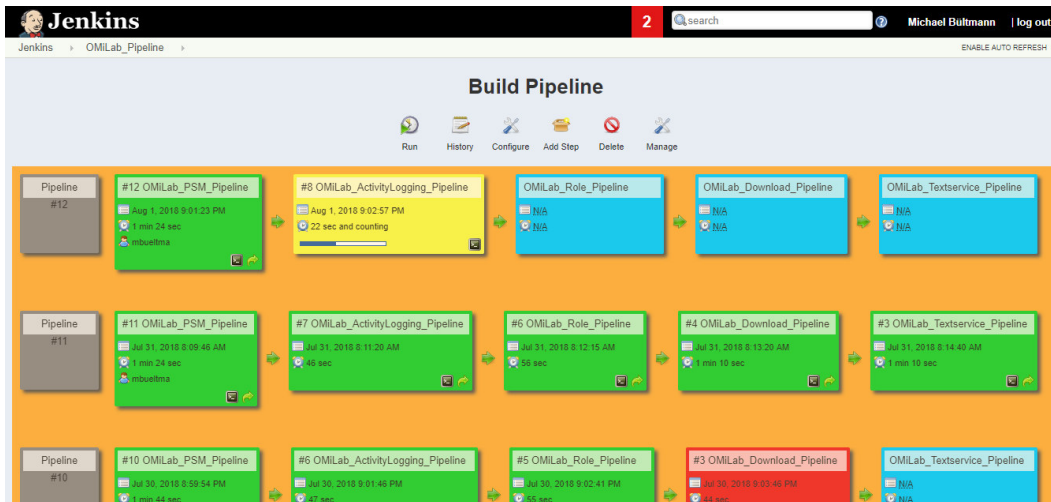


Figure 4.14: Jenkins: Build Pipeline

Further, there is the main dashboard with the list of jobs, the navigation bar and the currently running build jobs like in the figure 4.15.

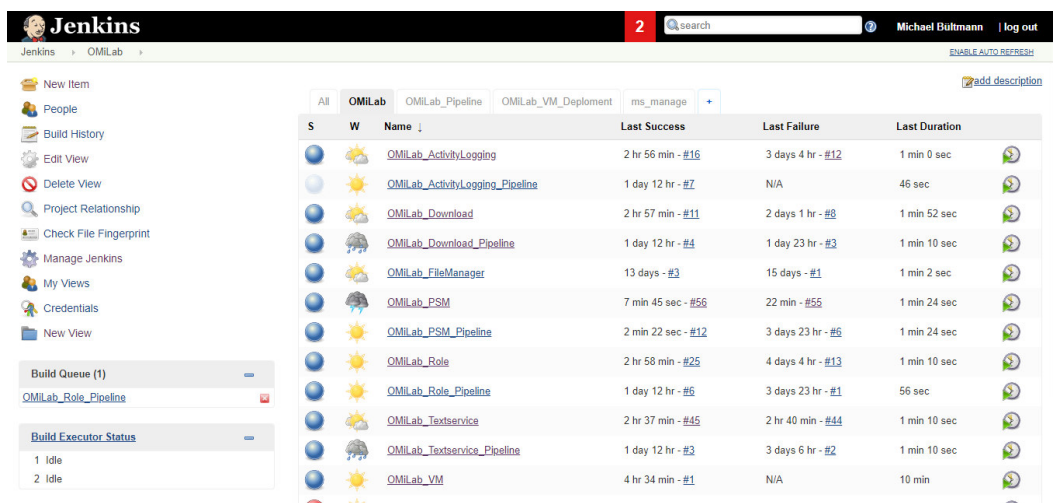


Figure 4.15: Jenkins: Dashboard

Other build server applications have a similar structure of configuration.

4 Evaluation

For example Bamboo¹⁹ structures its builds in plans and tasks containing settings for the source code and the build. Microsoft Team Foundation Server²⁰ works with tasks and pipeline processes as well.

4.3 Results

Figure 4.3 shows the final architecture of the OMiLab web application. The OMiLab architecture can be derived from the microservice architecture part 4.16 of the reference architecture figure.

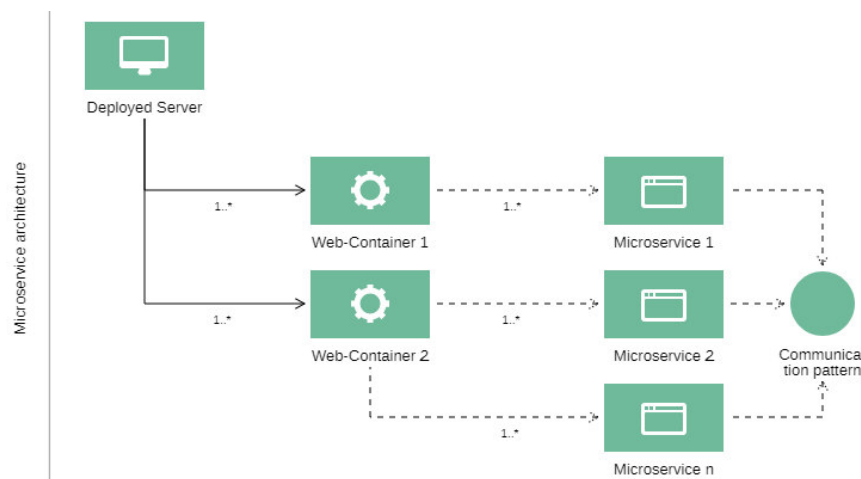


Figure 4.16: Reference software architecture: Microservice architecture

The overview of the OMiLab services focuses more on the interface, whereas the reference architecture emphasises on the location of the deployment. The reference architecture is more abstract regarding the interfaces and dependencies as it differs widely from project to project. The structure of a single microservice illustrated in figure 4.17, sums up the components of a microservice and is equivalent to the structure of the services in the

¹⁹<https://confluence.atlassian.com/bamboo/using-bamboo-289276852.html>

²⁰<https://www.visualstudio.com/de/team-services/continuous-integration> (Accessed on 23.05.2018)

OMiLab.

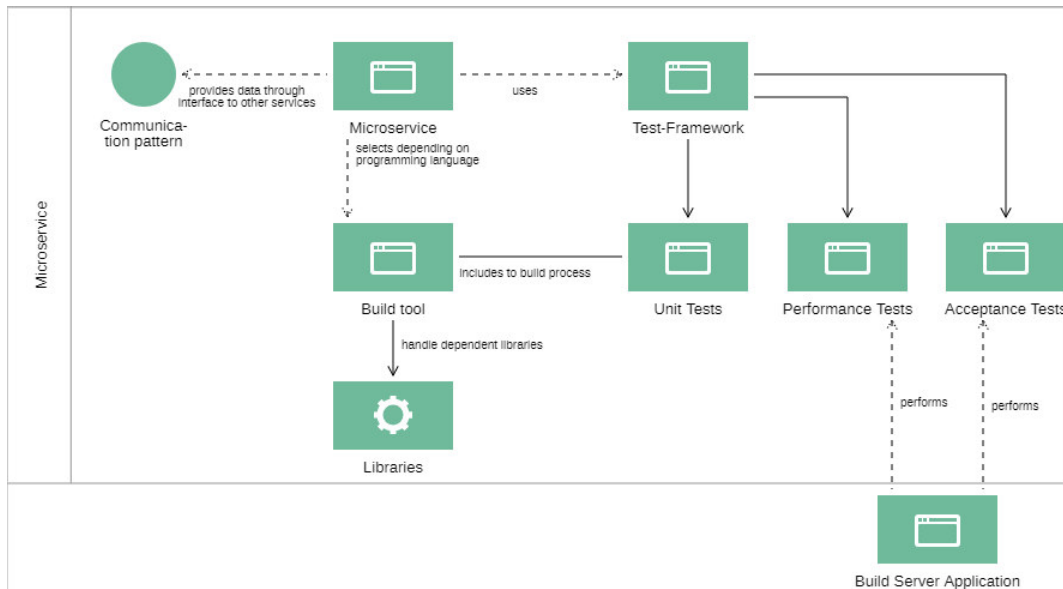


Figure 4.17: Reference software architecture: Microservice

The infrastructure in figure 4.7 can also be derived from the reference architecture 3.16. In the OMiLab the application for the continuous integration and the version control system are located on the same server. As both applications do not require permanently a high-performance this is a convenient solution.

The deployment is executed to a local server which is also listed as an alternative for the roll-out. The configuration in section 4.2.3 provides also a dynamic provision.

In general, the real-life example of the OMiLab web application fits the reference model well.

5 Discussion

As stated at the end of the introduction the manufacturing industry has to deal with the same challenges as the software development. Both domains must handle a higher complexity and react to changes in a shorter time. *Industry 4.0*, along with the related terms cyber-physical systems and the Internet of Things, tries to manage their processes with smart devices. These should be small, distributed and act independently as well as automated.

The challenges can be transferred to software development where the two following concepts can be pursued: microservice and continuous delivery. Microservice architecture provides an approach to handle the complexity in large applications and to equally support flexibility. This is achieved by small independent services working with their own technology stack. This independence allows interchangeability and easier scalability. For frequently used, complex web applications with high demands it is recommended to work with microservices to benefit from the advantages.

Continuous Delivery supports the automation of the build and the deployment of the applications. It evolves to a basic requirement to run a Continuous Delivery pipeline in the software development. It covers basic quality checks done by independent builds and tests. The complex deployments will be performed more often and therefore increase the reliability of productive installations compared to manual ones.

Both approaches help the development of applications significantly.

The chapter theory 2 presents an overview about the current research publications and gives an answer to the first research question regarding challenges and potentials of continuous delivery with a microservice architecture.

In consequence, the intention of the chapter 3 is the creation of a reference model which should cover most scenarios, abstracts them for further reuse

5 Discussion

and provides a first overview of the requirements for the integration. The chapter responds to the second research question about the definition of a reference model.

Afterwards the reference model needs to be evaluated with a practical example. The first application of the reference model to the environment of the OMiLab in chapter 4 went well. It is structured in microservices with only a few dependencies between each other and already provides basic continuous delivery functionality.

The SWOT-analysis in figure 5.1 presents the strengths, weaknesses, opportunities and threats. It mirrors the results of the first adaptation and answers also the third research question requiring the application to an existing scenario.

Strengths	Weaknesses
• Contains many required aspects of the integration of the continuous delivery and microservice architecture • Presents a good overview of all necessary components with phases and layers, which makes it easy to process • Good basis for future refinement	• Evaluated in only one project • Organisational effects and structure could not be reviewed effectively
Opportunities	Threats
• Elaboration of the reference model in more projects in the future • Refinement of the processes and infrastructure to fit more technology approaches	• New technology with a completely different approach

Figure 5.1: SWOT analysis of the reference model

For the integration of Continuous Delivery and microservices many aspects need to be considered. The reference model covers a great number of the required components. Continuous Delivery and microservice architecture have to fulfill many challenges. The pipeline of Continuous Delivery consists

of several components for each step. The implementation of a microservice architecture requires a well-planned process and structure. This is represented in the flow diagram and in the resulting architecture for the whole system as well as for a single service. The components of the OMiLab for example are mapped to the parts of the reference model.

The overview of the reference model helps the user to get a first insight of the single parts. The organization in phases and layers contributes to the understanding of the whole structure and makes it easy to proceed all processes in an ordered way. The processes can be executed step by step. Each task can be processed and corresponds to the artefact of the OMiLab environment. This applies to both main layers: software architecture and infrastructure.

The processes fit easily a certain level of abstraction. This provides a good basis for further refinement. All microservices of the example are implemented and organized in the same way as the reference architecture. It could also be applied to different technology stacks. The reference architecture and infrastructure are abstracted on a high level because they need to cover the structure of several different scenarios. The software architecture and infrastructure of the OMiLab can be related sufficiently to the reference architecture.

As the reference model is used in the context of only one project, it still needs to prove the usability for an adaptation. This point will be mentioned later under opportunities.

The development team of the OMiLab is rather small, so the organizational processes are not completely evaluated. This is an opportunity for an additional application in the future, which will involve a larger adaptation and more workers or teams assigned to it.

The current reference model is well structured and can be elaborated in more projects. Future projects can use this as a reference and provide feedback that should improve the whole reference model.

Especially new technologies not being used yet in the context of OMiLab can help to increase the robustness. The reference architecture and infrastructure will benefit from the new influences. This will either acknowledge and confirm existing implementation or add further useful details to it.

5 Discussion

There are not many threats to the reference model. It should be possible to apply it to several scenarios. Only a completely new technology with an entirely new approach of implementation and deployment, which can not be mapped to the reference model, could represent a threat.

Considering that the reference model has been applied for the first time it fulfills well its purpose as a recommendation of how to integrate the continuous delivery in a microservice architecture. It covers all major aspects.

Being used in future more often in different environments the reference model will be improved with more details.

6 Outlook

This chapter gives an outlook for further research of the topic continuous delivery and microservices.

In the future the importance of more flexible and scalable applications will increase. Therefore microservices are a good approach to manage the upcoming complexity in software systems (Dragoni et al., 2016). The advantages have been explained in detail in chapter 2.

The development of a microservice usually targets to a REST-API, which is supported by many programming languages and frameworks. Recent frameworks provide an easy approach to create a REST-API and this allows the developer to set-up a small and simple microservice in a couple of minutes¹.

On the other side it must be kept in mind that the system shifts more complexity from the local system to the network layer. Microservice architecture relies on a good management and monitoring of the services. There is always a potential to enhance these tools to facilitate the access to it (Dragoni et al., 2016).

Apart from the support of REST-API new frameworks try to increase the integrated functionality of the deployment. The npm² package manager for example supports the deployment in Node.js with its defined life-cycle relating to the build targets of maven and therefore works as an integrated build tool. The "package.json" file of the project offers the addition of commands to the life-cycle events like test and install.

For beginners it might be easier to understand the "package.json" of Node.js

¹<https://codementor.io/olatundegaruba/nodejs-restful-apis-in-10-minutes-qosgsfhbd> (Accessed on 22.06.2018)

²<https://npmjs.com> (Accessed on 22.06.2018)

6 Outlook

than the “pom.xml” of maven.

Regarding the challenges for Continuous Delivery Chen (2015a) mentions that there is no out-of-the-box solution available. There is only a limited number of tools and a high amount of own scripts are required. Some of recently released software suites try to come closer to an all-in-one solution by providing for example a version control system, a continuous integration server and an issue tracking system. Examples for these kinds of suites are GitLab³ or Microsofts Team Foundation Server⁴.

They evolve to out-of-the-box solutions but at the same time they decrease the flexibility of the technology. Therefore the software suite needs to support a variety of different approaches. Every software team has a different opinion of how to structure their application project. In the end it will be a major challenge to provide a working out-of-the-box solution containing all necessary parts of a microservice architecture with a running Continuous Delivery pipeline.

As the terms microservice architecture and Continuous Delivery become more popular, the companies will be more aware of it and will realise the demand for it.

The amount of documentation has increased which supports an easier access to the technology. There is more information and learning material available such as tutorials, courses and videos. The documentation covers all kinds of scenarios. Examples of specific documentation tools are Bamboo⁵ and Microsoft Team Foundation Server⁶.

For some complex systems the use of a microservice architecture is a “must-have”, because of the flexibility, the scalability and the possibility to migrate the legacy system step by step to a distributed structure (Wolff, 2015).

For the deployment it is highly recommended to use a Continuous Delivery pipeline. It guarantees automation and faster feedback (Wolff, 2014).

Both technologies have their challenges, but there is a high benefit regarding their strengths.

³<https://about.gitlab.com/product> (Accessed on 22.06.2018)

⁴<https://visualstudio.microsoft.com/de/tfs> (Accessed on 22.06.2018)

⁵<https://confluence.atlassian.com/bamboo> (Accessed on 22.06.2018)

⁶<https://docs.microsoft.com/en-us/vsts> (Accessed on 22.06.2018)

Appendix

Bibliography

- Botthof, Alfons (2015). *Zukunft der Arbeit in Industrie 4.0*. Berlin: Springer Vieweg. ISBN: 978-3-662-45914-0 (cit. on p. 10).
- Botthof, Alfons and Marc Bovenschulte (2009). "Das „Internet der Dinge“ Die Informatisierung der Arbeitswelt und des Alltags." In: *Hans-Böckler-Stiftung. Hans-Böckler-Stiftung. Düsseldorf (Arbeitspapier, 176)* 176 (cit. on pp. 7, 8).
- Brindescu, Caius et al. (2014). "How do centralized and distributed version control systems impact software changes?" In: *Proceedings of the 36th International Conference on Software Engineering*. ACM, pp. 322–333 (cit. on pp. 55, 56).
- Chen, Lianping (2015a). "Continuous Delivery: Huge Benefits, but Challenges Too." In: *Software, IEEE* 32.2, pp. 50–54 (cit. on pp. 28–30, 32, 33, 104).
- Chen, Lianping (2015b). "Towards Architecting for Continuous Delivery." In: *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*. IEEE, pp. 131–134 (cit. on pp. 34, 35).
- Claps, Gerry Gerard, Richard Berntsson Svensson, and Aybuke Aurum (2015). "On the journey to continuous deployment: Technical and social challenges along the way." In: *Information and Software Technology* 57, pp. 21–31 (cit. on pp. 37–39).
- Conway, Melvin E (1968). "How do committees invent." In: *Datamation* 14.4, pp. 28–31 (cit. on p. 25).
- Dragoni, Nicola et al. (2016). "Microservices: yesterday, today, and tomorrow." In: *arXiv preprint arXiv:1606.04036* (cit. on pp. 21, 23, 103).
- Fettke, Peter and Peter Loos (2004). "Referenzmodellierungsforschung." In: *Wirtschaftsinformatik* 46.5, pp. 331–340 (cit. on p. 18).
- Fettke, Peter and Peter Loos (2005). "Der Beitrag der Referenzmodellierung zum Business Engineering." In: *HMD-Praxis der Wirtschaftsinformatik* 241, pp. 18–26 (cit. on pp. 17, 18).

Bibliography

- Fill, Hans-Georg and Dimitris Karagiannis (2013). "On the conceptualisation of modelling methods using the ADOxx meta modelling platform." In: *Enterprise Modelling and Information Systems Architectures* 8.1, pp. 4–25 (cit. on p. 72).
- Fitzgerald, Brian and Klaas-Jan Stol (2014). "Continuous software engineering and beyond: trends and challenges." In: *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*. ACM, pp. 1–9 (cit. on pp. 39, 40).
- Fleisch, Elgar (2010). "What is the internet of things? An economic perspective." In: *Economics, Management and Financial Markets* 5.2, p. 125 (cit. on pp. 6, 7).
- Fowler, Martin (2007). "Mocks aren't stubs." In: Available on: <http://martinfowler.com/articles/mocksArentStubs.html> 6 (cit. on p. 58).
- Fowler, Martin and Matthew Foemmel (2006). "Continuous integration." In: *Thought-Works* [http://www.thoughtworks.com/Continuous Integration.pdf](http://www.thoughtworks.com/Continuous%20Integration.pdf) 122 (cit. on p. 63).
- Fowler, Martin and James Lewis (2014). "Microservices a definition of this new architectural term." In: URL: <http://martinfowler.com/articles/microservices.html> [accessed: 2016-10-31] (cit. on p. 27).
- Fowler, Martin and James Lewis (2015). "Microservices: Nur ein weiteres Konzept in der Softwarearchitektur oder mehr." In: *Objektspektrum* 1.2015, pp. 14–20 (cit. on pp. 23–28).
- Götzinger, David (2015). "Development of a Microservice based Franchise Infrastructure and Application to the Example of the Open Models Laboratory." Bachelor's Thesis. University of Vienna (cit. on pp. 74, 75).
- Götzinger, David, Elena-Teodora Miron, and Franz Staffel (2016). "OMiLAB: an open collaborative environment for modeling method engineering." In: *Domain-Specific Conceptual Modeling*. Springer, pp. 55–76 (cit. on p. 73).
- Griffor, Edward R et al. (2017). *Framework for Cyber-Physical Systems: Volume 1, Overview*. Tech. rep. 1500-201 (cit. on pp. 5, 6).
- Humble, Jez and David Farley (2010). *Continuous Delivery - Reliable Software Releases through Build, Test, and Deployment Automation*. Amsterdam: Pearson Education. ISBN: 978-0-321-67022-9 (cit. on pp. 61, 66).
- Kagermann, H. et al. (2012). *Im Fokus: das Zukunftsprojekt Industrie 4.0 ; Handlungsempfehlungen zur Umsetzung ; Bericht der Promotorengruppe Kommunikation*. Forschungsunion. URL: <https://books.google.at/books?id=v4qRjwEACAAJ> (cit. on p. 10).

- Karagiannis, Dimitris and Harald Kühn (2002). "Metamodelling platforms." In: *EC-Web*. Vol. 2455, p. 182 (cit. on pp. 71–73).
- Lasi, Heiner et al. (2014). "Industrie 4.0." German. In: *WIRTSCHAFTSINFORMATIK* 56.4, pp. 261–264. ISSN: 0937-6429. DOI: [10.1007/s11576-014-0424-4](https://doi.org/10.1007/s11576-014-0424-4). URL: <http://dx.doi.org/10.1007/s11576-014-0424-4> (cit. on pp. 1, 9, 10).
- Leppänen, Marko et al. (2015). "The highways and country roads to continuous deployment." In: *IEEE Software* 32.2, pp. 64–72 (cit. on pp. 36, 37).
- Louridas, Panagiotis (2005). "JUnit: unit testing and coiling in tandem." In: *IEEE Software* 22.4, pp. 12–15 (cit. on p. 58).
- Mulligan, Gavin and Denis Gračanin (2009). "A comparison of SOAP and REST implementations of a service based interaction independence middleware framework." In: *Winter Simulation Conference*. Winter Simulation Conference, pp. 1423–1432 (cit. on p. 46).
- Namiot, Dmitry and Manfred Sneps-Sneppé (2014). "On micro-services architecture." In: *International Journal of Open Information Technologies* 2.9, pp. 24–27 (cit. on pp. 21, 23, 24, 28).
- Olsson, Helena Holmström, Hiva Alahyari, and Jan Bosch (2012). "Climbing the "Stairway to Heaven"—A Multiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software." In: *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, pp. 392–399 (cit. on pp. 35, 36).
- O'reilly, Tim (2007). "What is Web 2.0: Design patterns and business models for the next generation of software." In: *Communications & strategies* 1, p. 17 (cit. on p. 6).
- Phillips, Andrew et al. (2014). *The IT Manager's Guide to Continuous Delivery: Delivering business value in hours, not months*. XebiaLabs (cit. on pp. 30, 33, 34).
- Pohl, K., G. Böckle, and F.J. van der Linden (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Berlin Heidelberg. ISBN: 9783540243724. URL: <https://books.google.de/books?id=jiTvwsMEgL8C> (cit. on pp. 10–12).
- Rajkumar, R. et al. (2010). "Cyber-physical systems: The next computing revolution." In: *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pp. 731–736 (cit. on pp. 3, 4).

Bibliography

- Rowson, Rob (2017). *2017 Version Control Software Comparison: SVN, Git, Mercurial*. URL: <https://biz30.timedoctor.com/git-mecurial-and-cvs-comparison-of-svn-software/> (visited on 10/12/2017) (cit. on pp. 54, 55).
- Schmid, Marco (2014). "Cyber-Physical Systems ganz konkret." In: *Economics, Management and Financial Markets* 7, pp. 62–64 (cit. on pp. 2, 3).
- Stachowiak, H. (1973). *Allgemeine Modelltheorie*. Springer-Verlag. ISBN: 9783211811061. URL: <https://books.google.at/books?id=DK-EAAAAIAAJ> (cit. on p. 17).
- Steinacker, Guido (2016). *Continuous Delivery von Microservices*. URL: <https://www.informatik-aktuell.de/entwicklung/methoden/continuous-deployment-von-microservices.html> (visited on 03/26/2017) (cit. on pp. 30, 32).
- Thönes, Johannes (2015). "Microservices." In: *Software, IEEE* 32.1, pp. 116–116 (cit. on pp. 22–24, 26).
- Vom Brocke, Jan (2015). *Referenzmodellierung: Gestaltung und Verteilung von Konstruktionsprozessen*. 2nd ed. Vol. 4. Logos Verlag, Berlin (cit. on p. 17).
- Wolf, Wayne (2009). "Cyber-physical Systems." In: *Computer* 42.3, pp. 88–89. ISSN: 0018-9162. DOI: <http://doi.ieeecomputersociety.org/10.1109/MC.2009.81> (cit. on p. 2).
- Wolff, Eberhard (2014). *Continuous Delivery: Der pragmatische Einstieg*. Heidelberg, Neckar: Dpunkt.Verlag GmbH. ISBN: 978-3-8649-0208-6 (cit. on pp. 29–31, 43, 57, 59, 60, 63, 64, 66, 67, 104).
- Wolff, Eberhard (2015). *Microservices: Grundlagen flexibler Softwarearchitekturen*. Heidelberg, Neckar: Dpunkt.Verlag GmbH. ISBN: 386-4-9031-30 (cit. on pp. 43, 45, 47, 104).