Sander Klock
3816958
s.klock@afas.nl
Utrecht University

Workload driven feature clustering to improve performance of a microservice architecture

Thesis - February, 2017

Supervisors
dr.ir. J.M.E.M. van der Werf (UU)
dr. S.L.R Jansen (UU)
J.P. Guelen MSc. (AFAS)

Universiteit Utrecht

If everyone is thinking alike, then somebody isn't thinking.

– GEORGE S. PATTON

Acknowledgements

This master thesis is the result of my graduation project of my Master Business Informatics (MBI) at Utrecht University. This thesis was created in cooperation with AFAS as part of the AMUSE project.

Many people have helped me with different aspects during this project. First of all, I would like to thank anyone that helped, advised or inspired me in any way related to my thesis. Several people put significant time and effort in helping me, and hence deserve to be thanked individually. First of all, I would like to thank my academic supervisors Jan Martijn van der Werf and Slinger Jansen for their help, time and feedback. Especially my bi-weekly discussions with Jan Martijn van der Werf resulted in many improvements to this thesis. My daily supervisor at AFAS, Jan Pieter Guelen, was of great help during the entire process. I would like to thank him for all brainstorm sessions, feedback and general guidance during my thesis process. Besides his normal work as software architect, he was always able to free some time for a discussion or brainstorm session, which is greatly appreciated. Furthermore Michiel Overeem's advices and feedback has proven to be valuable input that resulted in improvements in both the research and the created tooling. Jan Martijn van der Werf, Dennis Schunselaar and Guit-Jan Ridderbos were of great help during the creation and improvement of the formalisation of microservice architectures. Their experience and valuable insights have resulted in a better formalisation.

I specially want to thank my fellow students at AFAS: Marten Spoor, Pepijn Gramberg and Bart Smolders for all their feedback, suggestions, and for the many table football games. Furthermore, I want to thank all colleagues at the Architecture and Innovation department at AFAS for creating a great working environment and their help. I want to thank AFAS, and especially Rolf de Jong and Machiel de Graaf, for providing me with the opportunity to work at AFAS and the initial discussions that resulted in this research. Finally I want to thank my friends and family for their support and all non thesis related activities during the creation of my thesis.

Abstract

Interest in microservices architectures has increased over the last few years, with a significant increase since 2014. An increasing number of companies is evolving their architecture from a monolithic system to a microservice architecture. A microservice architecture provides more flexibility and scalability, at the cost of having a distributed system, eventual consistency and increased operational complexity. The impact of the advantages and disadvantages is defined by the size of individual microservices. Currently the size of a microservice is defined by metrics that are not related to performance and scalability. Since the large impact of the size of a microservice on the performance and scalability, metrics related to these quality attributes seem more appropriate than the existing metrics. The size of a microservice is defined by the features that it offers. As a result of these observations, this research aims to find an approach to optimize the performance of microservice architectures based on its workload. This research proposes an approach combined with accompanying proof-of-concept to alter a deployment to improve the performance. The proposed approach has been validated in a case study at AFAS, an ERP vendor in the Netherlands. This case study has validated that the approach works and has identified several interesting options for related research.

Contents

1 Introduction **6**
1.1 Problem statement . . . 7
1.2 Research questions . . . 8
1.3 Relevance . . . 9
1.3.1 Scientific contribution . . . 9
1.3.2 Societal contribution . . . 9
1.4 Thesis overview . . . 9

2 Research Approach **10**
2.1 Design Science . . . 10
2.2 Research design . . . 11
2.3 Problem statement . . . 12
2.4 Knowledge analysis . . . 12
2.4.1 Literature study . . . 12
2.5 Solution design . . . 13
2.6 Solution evaluation . . . 13
2.6.1 Case Study . . . 13

3 Running example **15**
3.1 Context description . . . 15

4 Modelling microservices **17**
4.1 Mathematical definition . . . 17
4.1.1 Deployment . . . 18
4.2 Graphical representation . . . 20
4.3 Technical representation . . . 21

5 Measuring the workload **23**
5.1 Defining a workload . . . 23
5.2 Obtaining the workload . . . 23
5.3 Measuring the workload . . . 24
5.4 Linking metrics to features . . . 26
5.5 Running example . . . 26

6 Deployment Fitness **28**
6.1 Queueing theory . . . 28
6.2 Software module clustering . . . 30
6.2.1 Fitness objectives . . . 30

7 Improving the deployment **32**
7.1 Optimization algorithm . . . 32
7.2 Genetic algorithm . . . 32
7.3 Genetic problem encoding . . . 34
7.4 Genetic operators . . . 36
7.4.1 Crossover operator . . . 37
7.4.2 Mutation operator . . . 39
7.5 Fitness calculation . . . 42
7.5.1 Queueing theory . . . 43

8 Case study **47**
8.1 Case study context . . . 47
8.1.1 Profit Next . . . 47
8.1.2 Microservice Architecture Model . . . 48
8.1.3 Workload . . . 48
8.1.4 Test setup . . . 49
8.1.5 Results . . . 52
8.1.6 Case Study Evaluation . . . 55

9 Discussion & Opportunities **57**
9.1 Findings and implications . . . 57
9.2 Validity . . . 57
9.2.1 Construct validity . . . 57
9.2.2 Internal validity . . . 58
9.2.3 External validity . . . 58
9.2.4 Reliability . . . 58
9.3 Limitations . . . 58
9.4 Opportunities . . . 59
9.4.1 Fitness objectives . . . 59
9.4.2 Different perspectives . . . 59
9.4.3 Self adapting software . . . 59

10 Conclusions **61**

Appendices **67**

A Case study results **68**
A.1 Low workload scenario . . . 68
A.2 Medium workload scenario . . . 69
A.3 High workload scenario . . . 69

1 | Introduction

Interest in microservice architectures has increased over the last few years, with a significant increase since 2014 according to Google Trends (2015) and Thoughtworks (2014). A microservice architecture is an architecture in which a single application is designed as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API (Fowler, 2014). As a result of this, every module is an independently deployable service. Combined with the lightweight communication protocols used, every service can use its own programming language and can be easily modified and scaled.

An increasing number of companies is evolving from a monolithic system to a microservice architecture, with well known examples like Groupon, Netflix and Amazon (Geitgey, 2013; Mauro, 2015; Fulton, 2015). These companies provide webservices to users all over the world. Traditional monolithic applications were unable to meet their scalability demands, since monoliths can only scale by horizontal duplication, running multiple identical copies of an application, or data partitioning, running multiple identical copies of an application that are each responsible for a subset of the data. In a microservice architecture, horizontal duplication and data partitioning can be applied to every individual microservice. This functional decomposition of the system in microservices (Abbott and Fisher, 2009) enables scaling of the system by only scaling parts of it.

A second reason why companies are moving to microservices, is the introduction of DevOps. DevOps is a way of working whereby developers and IT system operators work closely, collaboratively, and in harmony towards a common goal with little or no organizational barriers or boundaries between them (Swartout, 2014). Every DevOps team is responsible for both the development and operational aspects of their work, making it logical to map teams to microservices, since microservices are self-contained and can be redeployed individually. Introduction of Continuous integration or Continuous deployment is often combined with DevOps. Continuous integration (CI) means that software building and tested is automated; Continuous deployment (CD) expands CI by also automatically deploying software changes to production (Leppanen et al., 2015). Microservices provide a good platform to use CI or CD, since each microservice is self-contained, and can thus be easily build individually, tested and deployed.

However, microservices are no free lunch, as discussed in the popular article with the same title of the CTO of Contino (Wootton, 2014). Several difficulties introduced by a microservice architecture are related to it being a distributed system. Distributed

systems introduce additional points of failure and communication latency, since services need to communicate over the network. As described by the CAP theorem (Brewer, 2000, 2012), it is impossible to maintain strong consistency in a distributed system (Fowler, 2015) while maintaining availability and partition tolerance . Often a weakened form of consistency, called eventual consistency, is used. Vogels (2009) defined eventual consistency as a specific form of weak consistency; the storage system guarantees that if no new updates are made to the object, eventually all reads will return the last updated value. This introduces its own set of (usability) problems that need to be handled. Finally the operational complexity of a microservice architecture is higher than a monolithic application, since many services need to be managed, monitored and redeployed regularly and independently.

The impact of the advantages and disadvantages is defined by the size of individual microservices. Having many small microservices results in more communication between services, increasing networking overhead. Furthermore small services increase the operational complexity, since more services need to be managed.

The size of a microservice is directly defined by the features provided by the service. A microservice that offers more features will be larger than a service with only a few features. By moving features to other or new microservices, the performance and scalability of the system changes. The scalability increases, since the size of the smallest scalable unit becomes smaller. However, the effect on the performance of the system depends on the relationship between features. If, for example, two features are heavily dependent on each other, splitting them over different microservices might result in significant communication overhead, resulting in a decrease of performance. Merging two microservices results in a loss of scalability, but performance might increase due to decreased communication overhead. Secondly the actual usage of features by users determines the impact of moving features. If a seldom used feature is moved, the impact is much smaller than moving a frequently used feature.

1.1 Problem statement

The term microservice indicates that services should be small. However, people are reluctant to define how small they should be (Stenberg, 2014). There are several metrics (of varying quality) for the size of microservices, such as lines of codes, being able to rewrite a microservice in 6 weeks or having a 2-pizza team (two pizzas are enough to feed the entire team) per service (Fowler, 2014). Another typical answer is that a microservice should do one thing, which leaves room for interpretation.

None of the existing metrics are related to quality attributes, but as stated in the introduction, the size of a microservices has a direct impact on the performance and scalability of the application. As a result of this observation, metrics related to performance and scalability seem more appropriate than the existing metrics.

Currently there exists no framework to optimally size microservices based on performance and scalability. This thesis aims to fill this gap in scientifc knowledge. To achieve this, two subproblems need to be addressed. First, relevant metrics for measuring the performance and scalability of a microservice architecture

have to be determined.

Based on these metrics, the second subproblem can be addressed, namely the creation of an analysis method to optimally size microservices. Based on this analysis, improvements should be deducible. The suggested improvements will be based on the three observations described in the introduction:

- The number of features determines the size of a microservice.
- Placement of features in microservices has an impact on the scalability and performance
- Actual usage of features influences the impact of placement of these features on the system's performance and scalability.

Using these suggestions for improvement, the architect will be able to optimally size the microservices with regards to performance and scalability for the actual usage of the application.

1.2 Research questions

Based on the problem statement with related observations, the following research question is defined:

RQ | *How to improve performance of a microservice architecture by clustering features based on application workload?*

The first subproblem mentioned in the problem statement, determining which metrics are relevant to measure the performance and scalability of a microservice architecture, resulted in the following subquestion:

SRQ 1 | *Which metrics are relevant for clustering features in microservice architectures?*

By answering this subquestion, the first problem is addressed. The goal of the second subquestion is to link the obtained relevant metrics to the features that will be analyzed as part of the second subproblem.

SRQ 2 | *How can metrics be linked to features?*

The third subquestion aims to find an efficient way to find performance problems based on the metrics provided. Based on the linkage between the metrics and features, the second part of this subquestion consists of finding more efficient feature groupings by moving individual features.

SRQ 3 | *What is an effective way to find improvements in the grouping of features in a microservice architecture?*

Finally, the goal of the last subquestion is to evaluate the effect of the modifications with regards to performance and scalability.

SRQ 4 | *What is the effect of the architecture modifications on scalability and performance?*

1.3 Relevance

This section will address both the scientific and the societal contribution of this thesis.

1.3.1 Scientific contribution

As stated in the problem statement, there exist no framework or method to optimally size microservices based on performance and scalability. The goal of this thesis is to fill this gap in scientific knowledge. Secondly it will expand the published scientific material on microservice architectures, which is still limited. This thesis combines several fields of research such as Microservices, Performance engineering, and Performance analysis based on traces. Finally this thesis contributes to the AMUSE Project[1].

1.3.2 Societal contribution

The created framework can be used by organizations to improve the scalability and performance of their microservice architectures. By optimizing the performance, organizations can use their hardware more efficiently and thus reduce software operation costs and potentially reduce their energy consumption.

1.4 Thesis overview

At first the research approach is discussed in chapter 2. Afterwards chapter 3 introduces the running example that is used as guiding example in this thesis. Chapter 4 introduces a formal, graphical and derived technical representation of a microservice architecture. In the next chapter, the other major component of this research is addressed, measuring the workload. Chapter 6 describes the function that is used to determine the quality of a deployment. This is followed by a description of the modification operators on a deployment and the approach to improve the deployment for a given workload in chapter 7. The next chapter describes the case study performed at AFAS. This is followed by a discussion of this research in chapter 9. Finally chapter 10 will describe the conclusions of this research.

[1] https://amuse-project.org/

2 | Research Approach

This chapter describes the used research methods to conduct this thesis research. As part of this thesis a literature study and a case study are performed.

2.1 Design Science

Hevner et al. (2004) describe the two main paradigms that characterize much of the research performed in the Information Systems field:

Behavioural science has as goal to create and verify theories that explain or predict human or organizational behavior

Design science creates new and innovative artifacts with the goal of extending human and organizational capabilities.

Since the goal of this thesis is to create new artifacts that help with optimizing of microservice architectures based on a given workload, this thesis falls in the category of Design science. Figure 2.1 shows this thesis applied to the Information system research framework by Hevner et al. (2004).

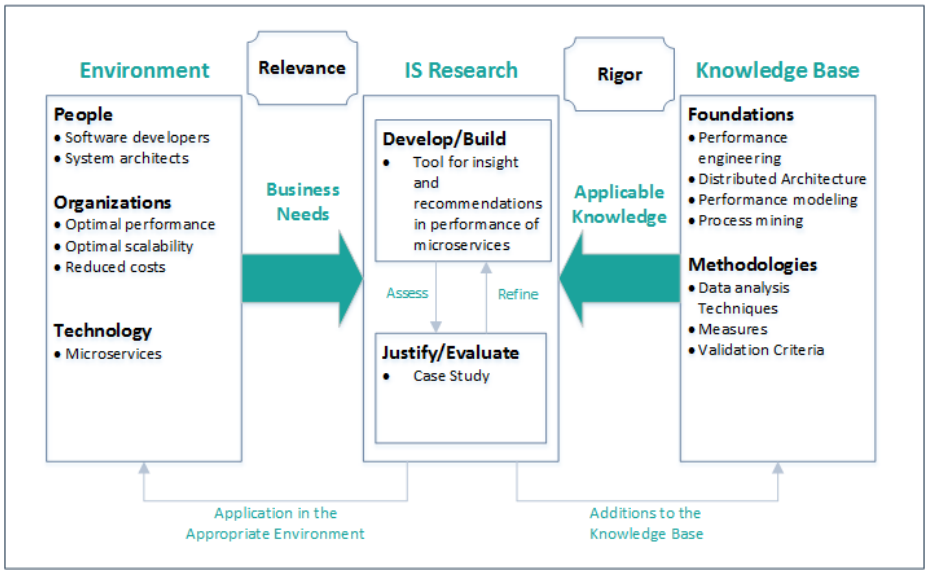


Figure 2.1: This thesis projected on the Information system research framework (Hevner et al., 2004)

2.2 Research design

The following research phases have been derived from the Information system research framework by Hevner et al. (2004) combined with the research cycle of Polya (2014):

Problem statement The goal of the first phase is to describe, understand and explain the problem. The deliverable of this phase is the problem statement.

Knowledge analysis This phase aims to identify the available scientific knowledge related to this problem. An overview of related literature is delivered as a result of this phase.

Solution design In this phase the current scientific knowledge is extended by designing candidate solutions for the problem. The second sub phase consists of selecting the best candidate solution for the problem at hand.

Case study implementation The best candidate solution is implemented as part of a case study in this phase.

Solution evaluation The final phase evaluates how well the case study implementation solves the initial problem.

Figure 2.2 shows the phases and their information flow. The problem statement phase is related to the Business Needs in the model of Hevner et al. (2004), since the Business Needs result in the problem statement of this research. Secondly, the knowledge analysis phase is related to the Knowledge Base and the related Applicable Knowledge flow in the model of Hevner et al. (2004). The Develop/Build phase of Hevner's model is covered by the Solution Design phase of this thesis. The last two phases of this research, Case study implementation and Solution evaluation, are related to the Justify/Evaluate phase of the framework by Hevner et al. (2004).

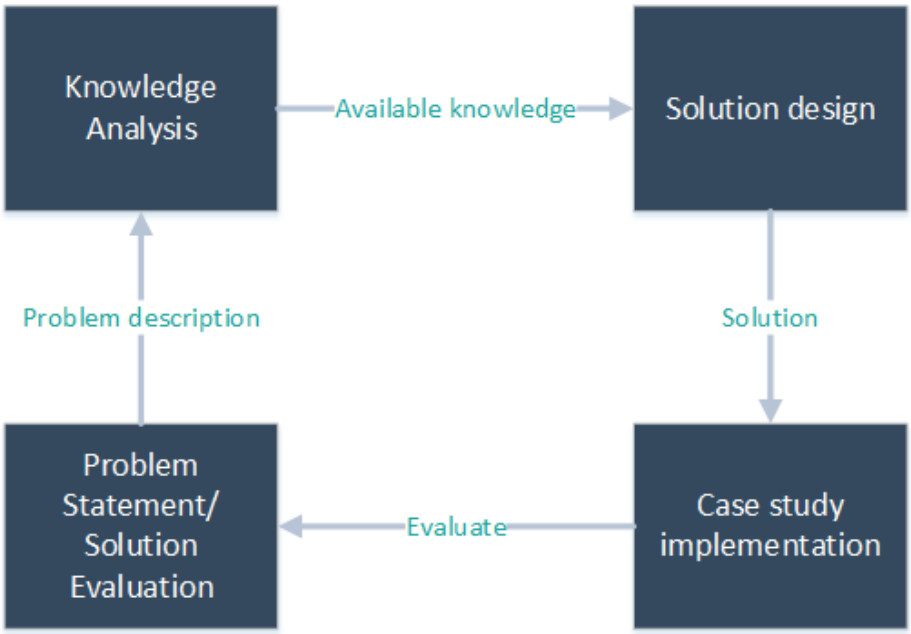


Figure 2.2: Research cycle

2.3 Problem statement

The goal of this phase is to describe, understand and explain the problem. This is achieved by performing an exploratory related literature study and exploring the problem at AFAS. The exploratory related literature study is performed using the snowballing approach. Scientific search engines such as Google Scholar and DBLP are used to search for papers based on keywords related to the problem statement. Secondly popular blogs of industry experts will be searched for information on the topic. This phase resulted in the project proposal.

2.4 Knowledge analysis

The goal of this phase is to find applicable scientific knowledge to the problem. Based on the problem description, relevant literature will be studied.

2.4.1 Literature study

The performed literature study consists of two phases. The first phase is an exploratory study, in which relevant areas of research are identified and introductory papers are studied. For this phase an unstructured literature study, using the snowballing approach is used for every relevant area of research. This method is chosen since the relevant literature is scattered in different areas of research. This phase identified the following main relevant areas of research:

- Microservice Architectures
- Software Scalabilty
- Software Performance Engineering
- Trace based performance analysis
- Software module clustering
- Self-adapting software systems

The second phase continuous on the relevant areas of research identified in the first phase. For each area an unstructured snowballing approach is used to obtain relevant papers. The main goal of this phase is to gain a more in-depth insight in the field of research. However, some relevant papers were identified that are (partially) located in other areas of research.

2.5 Solution design

The goal of this phase is to create a solution for the problem at hand based on existing knowledge. The deliverable of this phase is a tool.

2.6 Solution evaluation

The goal of this phase is to evaluate the effect of the architecture modifications on the performance and scalability of the implemented solution at AFAS.

The solution created in the Solution design phase will be used to create a new microservice architecture. The same workload simulations will then be ran on this modified architecture. Based on the results of this phase, SRQ 5 will be answered. The results of this phase will be used to improve the process and deliverables of the previous phases, as well as produce an evaluation.

2.6.1 Case Study

A case study will be performed to evaluate the effectiveness of the created solution. The case study will be performed at AFAS. The details of the case study context are described in chapter 8. A specific form of a case study is performed, namely a controlled experiment involving an intervention and outcome of a single system (Yin, 2013). As a result of time constraints, only a single case study can be performed. AFAS's software provides an excellent platform to test the created framework, as described in the case study context. Additionally this thesis aligns with the goals of the broader AMUSE project, which is a research collaboration between AFAS, Utrecht University and VU University Amsterdam. This makes a case study at AFAS a logical decision.

Single case studies typically have a decreased generalizability and reproducibility. Several measurements have been taken to increase the validity of this case study.

External validity

Open standards and open-source components will be used in order to ease implementation in a different microservice environment, increasing the reproducibility.

The created application will consist of an AFAS specific input module that transforms their application logs to the standard format required by the analyzer and visualizer. The analyzer and visualization component will be made open-source and published on Github.com, making it easier for companies and researchers to reproduce it with their environment.

Internal validity

To increase internal validity, insight will be provided in the data collection method. Thorough insight in the analysis of the gathered data will be provided, so the process can be reproduced and reviewed. Finally the criteria that will be used to compare the modified architecture with the original architecture (control group) will be clearly reported.

Construct validity

To improve construct validity, existing definitions of measures are used whenever available in literature. If needed, existing definitions will be extended. Additionally, the rationale behind an extension of a definition will be provided.

Reliability

The case study protocol will be documented. The goal of this documentation is to describe the procedures so that an auditor could follow them and reproduce the same results. Since a large part of the method will be automated, it should be relatively easy for an auditor to reproduce the same results.

3 | Running example

This chapter presents a running example that will be used for clarification in the rest of this thesis. The running example will provide a minimal case in which the benefits of this research are clear.

3.1 Context description

Webshop.com is a large web shop with millions of customers and many concurrent orders. They are currently making the transition to a microservice architecture. Being a traditional web shop, customers can browse Webshop.com's catalog, view products, add products to their shopping cart, and checkout their shopping cart.

The software architects of Webshop.com are wondering how to distribute these functionalities of the monolith over the microservices. The provided features are described below:

Customer This feature manages the accounts of all customers of the web shop. It contains common customer data such as an email address, password and (default) shipping details such as zip code, city and street name.

Product This feature contains product information such as description, images, price and other product information.

Order The order feature enables customers to put products in their virtual shopping cart. Secondly this feature enables customers to view their order history.

Payment This feature enables customers to checkout their shopping cart.

Delivery This feature tracks the status of a delivery starting from a paid shopping cart till the actual delivery of the order.

Review The review feature enables customers to post a review of product.

There are several dependencies between those features, as depicted in figure 3.1. The customer and product features have no dependencies. The review feature requires the data of the customer that created the review and the product that the review is about. The order feature of the web shop requires both a reference to the customer that made this order, and a set of products that are ordered. Finally both the payment and delivery features need information of the order.

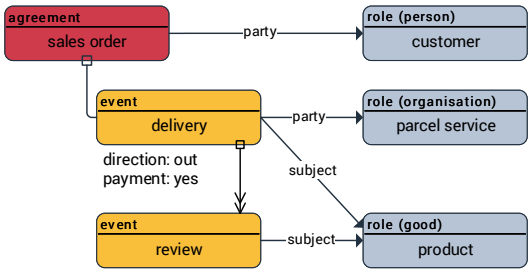


Figure 3.1: Model of the features of Webshop.com

The software architects foresee several actions in which transactions over multiple features occur, that are potentially located in different microservices. To tackle this distributed data management problem, they decided to create an event driven microservice architecture. An event-driven microservice architecture is an architecture in which feature publish an event whenever it updates its data, which can be subscribed upon by other features located in different microservices. These subscribed features can then update their data based on the event (Richardson, 2014). This however results in an eventually consistent system.

4 | Modelling microservices

This chapter describes a mathematical description of a microservice architecture. Furthermore both the graphical and technical representation of this model are discussed.

4.1 Mathematical definition

The basis of a microservice model is a set of features F that needs to be distributed over the microservices. This distribution is influenced by the dependency graph of the features, called R, since dependencies will typically be placed in the same microservice. Secondly a feature is defined as a set of properties. As a result, the set of features F is a partitioning of the set of properties P. This means that every feature consists of a disjoint subset of P and that the union of all features is P. This results in the following formal definition of a feature model:

Definition 4.1.1 (Feature model)
A feature model is a 3-tuple (P, F, R) with

- a set of *properties* P;
- a set of *features* F, being a partitioning of P;
- and the *dependency graph* (P, R), a directed graph.

Note that (P, R) is a dependency graph that can contain cycles. Since we consider the dependency satisfied if the dependant property is located in the same microservice, cyclic dependency graphs are not a problem for our approach.

To make this definition more concrete, it will be applied to a part of the running example described in chapter 3. For brevity only a part of the running example is formalised. Three features will be formalised: the sales order (O), the customer (C), and the delivery feature (D).

In order to keep the model small, each feature only has the bare minimum of properties. The order has an identifier (P_1), a reference to the customer (P_2), and products (P_3). The customer has an identifier (P_4), address (P_5), and name (P_6). Finally the delivery component has a reference to the order (P_7), reference to the customer (P_8), and delivery address (P_9). The delivery address can be a reference to the address of the customer, or it can be a manually entered alternative address.

This model is depicted in figure 4.1.

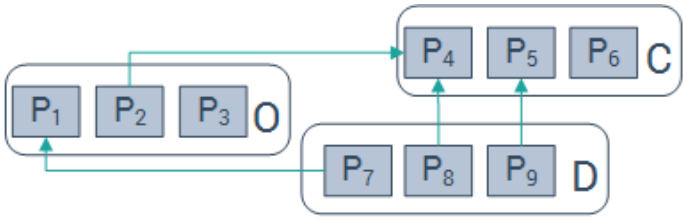


Figure 4.1: Feature model of a part of the running example

This example can be represented by a feature model. At first the set of features F will be described. The example states that there are three features (O,C and D), this is described by: $F = \{O, C, D\}$. Since the definition of a feature states that a feature is defined as a set of properties and the example states that each feature has three properties, F can be represented as $\{\{P_1, P_2, P_3\}, \{P_4, P_5, P_6\}, \{P_7, P_8, P_9\}\}$. The properties ($P$) of the feature model are defined as the set of all properties in the system. Definition 4.1.1 states F is a partition of P. By using the property that the union of all sets in F is P, this results in $P = \{P_1, P_2, .., P_9\}$. The last item of a feature model is the set of directed edges that represent the dependencies between properties. This is described by R using $(property, dependency)$ pairs, so in our example this would result in $R = \{(P_2, P_4), (P_7, P_1), (P_8, P_4), (P_9, P_5)\}$.

4.1.1 Deployment

A microservice architecture is an instantiation of a feature model such that every microservice contains one or more features with one or more properties. There exists a special instantiation of every feature, that instantiates all properties of that feature.

In event-driven microservice architectures the logic needed to handle the events has to be duplicated to different microservices. We consider this duplicated features. These duplicated features are however not exposed publicly, but are only available to features inside the microservice. This implies that a feature can have multiple instances in a deployment. Since a feature model does not allow a feature to appear more than once, the concept of feature instances I is introduced. Each feature instance i can either be public or internal. Additionally an internal feature might contain a subset of the properties of the public feature. A common example is to only include the identifier and name of an referenced entity, instead of the entire entity.

Definition 4.1.2 (Microservice Architecture)
Given a feature model (P, F, R), a *Microservice Architecture* is a 4-tuple (I, M, λ, h) with:

- a set of *feature instances* I;
- a set of *microservices* M, being a partitioning of I;
- the *property instantiation function* $\lambda : I \to \mathbb{P}(P)$, a total function that maps each feature instance to a set of properties;
- the *public instance function* $h : F \to I$, a total function that defines for each feature its public instance;

such that

- Each microservice contains all instances necessary to fulfil the dependency requirements, i.e.

  $$\forall m \in M\colon \forall i \in m, p \in \lambda(i), q \in P : \\ (p, q) \in R \implies \exists j \in m : q \in \lambda(j)$$

- Every microservice contains each feature at most once, i.e.

  $$\forall m \in M\colon \forall i, j \in m : \exists f \in F : \\ (\ \lambda(i) \subseteq f \wedge \lambda(j) \subseteq f\ ) \implies i = j$$

- Each feature instance is a subset of its feature, i.e.
  $\forall i \in I : \exists f \in F : \lambda(i) \subseteq f$
- Each feature has a public instance that is equal to itself, i.e.,
  $\forall f \in F, i \in I : h(f) = i \implies f = \lambda(i)$
- Each microservice contains at least one public feature instance, i.e.
  $\forall m \in M : \exists i \in m, f \in F : \lambda(i) = h(f)$

This definition will be applied to the part of the running example presented earlier in this section. This example will describe the deployment in which every feature will have its own microservice. Hence there will be three microservices (O,C,D). As a result of the dependencies defined in the feature model, this introduces internal feature instances in the order and delivery microservice. A public feature instance is denoted by i_X, with X being the name of the feature. Internal feature instances with a subset of the properties are denoted by $i_{X\{1,2,3\}}$ for a feature instance of X with three properties (1,2,3). An internal feature instance with all properties is denoted by $i_{X\{1..n\}}$.

For the order microservice (O) this will result in the following instances: $m_O = \{i_O, i_{C\{4\}}\}$. The customer microservice (C) can be represented as $m_C = \{i_C\}$ and

the delivery microservice (D) as $m_D = \{i_D, i_{O\{1\}}, i_{C\{4,5\}}\}$. Hence the entire deployment with all public features in a separate microservice can be represented as $D = \{m_O, m_C, m_D\}$.

A deployment in which all public feature instances are located in the same microservice, results in a simple deployment: $D = \{m_{OCD}\}$ where $m_{OCD} = \{i_O, i_C, i_D\}$.

A deployment in which the delivery feature and the customer feature are grouped together in a single microservice, and the order feature in another microservice results in the addition of two internal feature instances as a result of the dependencies. An internal instance of the customer feature in the order microservice has to be added as a result of P_2 depending on P_4. Secondly P_7 depends on P_1, hence an internal order feature has to be placed in the customer and delivery microservice. This deployment can thus be described by: $D = \{m_{CD}, m_O\}$ where $m_{CD} = \{i_C, i_D, i_{O\{1\}}\}$ and $m_O = \{i_O, i_{C\{4\}}\}$.

4.2 Graphical representation

While the formal definition of the model has its benefits, a graphical representation is easier to understand and communicate with other stakeholders. Hence this section introduces a simple visualization that is designed for these use cases. It should be noted that creating this visualization is not the focus of this thesis, hence it can be improved by future research. This visualization is purely based on the experiences of the author during his thesis. Figure 4.2 provides the graphical representation of the deployment discussed above in which every feature is placed in a separate microservice. It is important to note that the colours used can be adjusted to the context. The visualised deployment contains three microservices, depicted by the hexagons.

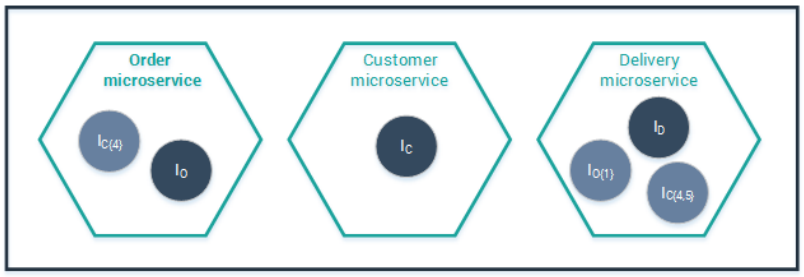


Figure 4.2: Deploymentmodel graphical representation

The darker circles inside a microservice represent publicly exposed features of a microservice, while the lighter circles represent an internal feature. It is important to note that properties are not displayed in this representation. If the properties would have been included for the features, the figure would become very large and crowded. However, any user is free to depict properties by placing elements inside the feature, but it is not recommended. Secondly the relations are not explicitly displayed in this representation, since typically features have many relations, resulting in many arrows

in the visualization. Again a user is free to depict relations as an overlay over this representation, however it is recommended to use a different type of diagram such as a chord diagram or graph to depict these relations.

4.3 Technical representation

A technical representation of the above model has been implemented in the tool accompanying this thesis. It uses JSON as transport format. JSON is a lightweight data-interchange format that is easy to read and write for both humans and computers (Crockford, 2006). Below the structure of the created model is described.

The model, called a deployment model, is expressed as an object with three top-level attributes as shown in Listing 4.1

```
{
        "version" : integer,
        "microservices": [{..}],
        "relations": [{..}]
}
```

Listing 4.1: Deployment structure

The version attribute represents the version, using an integer, of this particular deployment model. This attribute should be used by parsers to check whether they are able to parse the contents of the model. The microservices attribute contains an array of microservice objects, that will be discussed next. As shown by the mathematical model, a deployment is essentially a set of microservices, hence an array is used. Finally the relations attribute contains an array of relation objects, that will be discussed below. Again the relations are a set of individual relations, making an array representation a logical choice.

Every microservice in the microservices array has the structure shown in listing 4.2.

```
{
        "id" : "string",
        "name": "string",
        "features": [{..}]
}
```

Listing 4.2: Microservice structure

A microservice has an id attribute, representing its unique identifier. This identifier represents a microservice. The name attribute contains a user friendly name of the microservice, which is not guaranteed to be unique. The last attribute, features, is an array of all feature objects located in this microservice.

A feature object corresponds to a feature instance in the mathematical model described above. It consists of the attributes described in listing 4.3.

```
{
```

```
        "id" : "string",
        "internal" : boolean,
        "name": "string",
        "properties": [{..}]
}
```

Listing 4.3: Feature structure

The id attribute of this feature represents which feature this feature instance describes. Hence a feature id is not guaranteed to be unique. A feature that has been duplicated in another microservice must have the same id attribute as the main public feature. The internal attribute defines if a feature is publicly exposed (false) or internal (true). The name of a feature is a user friendly name, which is not guaranteed to be unique. The properties attribute is an array of all properties of this feature.

A property is represented by the structure that is shown in listing 4.4. It contains an id attribute that represents the property in the feature model. The name of a property represents its non-unique user friendly name. Finally the weight represents the weight that indicates how much processing time is required for this property. For example a complex calculated field might have a higher weight than a simple boolean.

```
{
        "id" : "string",
        "name": "string",
        "weight": integer
}
```

Listing 4.4: Property structure

Finally the relations array of a deployment model consists of relation objects, of which the structure is shown in listing 4.5. A relation contains two attributes, an identifier of the source feature, and the identifier of the target of the relation.

```
{
        "sourceId" : "string",
        "targetId": "string"
}
```

Listing 4.5: Property structure

5 | Measuring the workload

As stated in the problem statement, performance and scalability related metrics are required to optimally size microservices. This chapter first defines the concept of workload, followed by a discussion on methods to obtain the workload. Afterwards the link between workload and performance together with the scalability of a system is discussed. Finally the insights obtained in this chapter are applied to the Webshop.com running example.

5.1 Defining a workload

Before the architecture of an application can be optimized based on the workload of the application, the concept of workload should be clear. Intuitively the concept of workload is "how the application is used by its users". This raises the question who the users of the application are. Two main groups of users can be distinguished: human end-users that interact with the application using the user interface and other software systems that interact with the application (API). Furthermore time is an important dimension in the usage of an application. The usage of features over time provides insight in patterns, that might repeat often and are worth optimizing for. Furthermore the usage over time gives insight in peak usage, while this insight is lost if aggregations, such as an average, are used.

As a result of the previous observations, workload is defined in terms of concurrent users and used features as a function of time.

5.2 Obtaining the workload

By monitoring the operations of a system, its workload can be derived. The data obtained from monitoring the operation of the system is collectively called Software Operation Data (van der Schuur et al., 2011; van der Werf and Verbeek, 2015). While Software Operation Data encompasses all data obtained by monitoring a running system, we are only interested in the usage of features as a result of the definition in the previous section. The previous section also showed that applications are used by two main user categories: humans and other software systems. Since other software systems typically use an API of the application instead of the user interface, measuring the interactions with the user interface is not sufficient in general.

In case of a client-server-model application, a model where clients interact by requesting services of servers, which provide a set of services (Bass et al., 2007), it is possible to measure the usage of features at the server. A typical example of a client-server-model is the retrieval of a website by a web browser from a web server. Most webservers have an access log, which logs a timestamp and request information. Both requests made by humans using their web browser and other software systems are present in this log, making it a good starting point to extract the workload. These type of logs are also frequently used in process discovery process mining. Since microservice architectures are typically used at the server part of the client-server-model, a good starting point for extracting the workload are these access logs. Process mining (Aalst, 2012), a technique that aims to discover, monitor and improve real processes by extracting knowledge from event logs, is a technique that can be used to extract user behaviour over time from the access logs.

5.3 Measuring the workload

While for example the access logs mentioned in the previous section provide insight in the usage of the feature, they do not provide insight in the performance and scalability of the system. Hence different Software Operation Data is required to obtain these insights, such as metrics.

Metrics are commonly used to monitor a running software system. However, many definitions of a metric exist. We define a metric as a more abstract representation, such as an average or sum, of a time series of individual measurements. A measurement is a quantitative attribute of a running software system that can be measured automatically. An example of a metric is average response time, the average time it takes the system to respond to a request. The measurements in this example would be the individual response times.

The following different levels of metrics, based on Meier et al. (2004), can be identified:

Application metrics Application metrics are metrics reported by the application itself. These metrics are specific for the application and are typically related to significant events in the domain of the application. An example of such a metric might be the number of products ordered per minute for a web shop.

Platform metrics Platform metrics are metrics reported by the framework or runtime of the application. These metrics are related to significant events occurring in the runtime of the application. For example the Microsoft .Net platform reports the number of exceptions thrown per second (Meier et al., 2004).

System metrics System metrics are reported by the operating system and/or hardware of the server. These metrics provide information about significant events on the hardware level. The number of CPU interrupts per second is an example of a system metric.

Application metrics require the most effort of an application developer, since he needs to implement the calculation and reporting of the metrics. However these metrics provide relevant insights for the business as well. Depending on the framework or platform on which the application is build, several metrics might be available that provide insight in technical reasons for performance problems, such as the frequency of garbage collections. Finally system metrics are commonly used by system administrators to monitor software and the hardware, since they are easy to collect and they indicate direct resource bottlenecks.

When a system is throughput-limited, but none of the system resources (CPU, memory etc.) is saturated, another resource is the bottleneck of the system (Woodside et al., 2007). A common example of such a bottleneck is database lock contention (Cecchet et al., 2003). With database lock contention, many concurrent writes to a single record or table are attempted, resulting in some threads or processes having to wait until the lock is released. This waiting state does not require significant CPU or memory usage, hence it is not detectable using the system metrics. This type of bottleneck can be detected by monitoring the metrics provided by the database engine itself, or by application metrics that monitor the duration of database queries.

Within an application, runtime metrics can indicate abnormalities or problems. However, such a metric does not necessarily provides insight in the cause of the abnormality. For example the response time of a HTTP request might increase significantly. This metric does not provide insight in the cause of the response time increase, hence in this case the response time is considered a symptom metric. The increased response time might for example be caused by higher latencies or increased database query times. The metrics that indicate the core cause of the abnormality or problem are called causal metrics (Rob Ewaschuk, 2016).

Monitoring the performance of a distributed system, such as a microservice architecture, is a complex task (Braddock et al., 1992). A (small) performance problem within a single microservice will cascade in reduced performance of other microservices that depend on that microservice. With monolithic applications, one can typically attach a profiler to the running system and identify the cause of the performance problem. With a distributed system, one has to attach a profiler to each component of the running system, which becomes impossible in practice.

Typically metrics are used to provide insight in the performance of a distributed system. This requires a combination of application, platform and system metrics. Depending on the granularity of the application metrics, these metrics mainly serve as symptom metrics or might also provide insight in the root cause of the problem. The platform metrics often provide insight in increased error rates in the application itself. The system metrics act both as causal and symptom metrics. System metrics might for example correctly indicate a saturated network link as cause of performance problems, but in another case high CPU usage might be a result of an increased request rate.

As shown by the previous examples, it is important to have an holistic view of the performance of the system, since most metrics need context in order to be valuable. It is common practice to add unique request identifiers to incoming requests to achieve traceability through the entire system. Using these traces, it is possible to obtain a

(performance) stack trace of the request through the system to identify the root cause of a performance problem.

5.4 Linking metrics to features

Platform and system metrics generally have process level as smallest granularity level. This means that common metrics such as memory usage are available for individual processes. A finer level of detail can be achieved using profilers, however they have a significant impact on the performance of an application, making them not suitable for production environments. While process level granularity typically provides sufficient details for monitoring a running application, it does not provide sufficient detail to link them to individual features, since every processes contains one or multiple features.

Depending on the type of platform metric, it might be possible to link them to individual features using feature location techniques (Rubin and Chechik, 2013; Dit et al., 2013). Since the application developer has full control over the application metrics, it is possible to link them to individual features in the system. Typically these metrics are linked to features by adding metadata to the metric. It is recommended to use a logging and monitoring system that support structured metrics, in order to support metadata. If the metadata contains both a unique request identifier and feature identifying data, the performance impact of the system's usage per feature can be determined.

5.5 Running example

First Webshop.com needs to extract their workload. Since their application is provided as a web application, the HTTP access logs of their microservices provide a good starting point. These access logs will not provide the full picture, because the microservices can exchange messages using a message bus. These messages should also be incorporated in the workload, and they should also be linked to the original HTTP request that resulted in the message. This requires some kind of request end-to-end traceability. Secondly they need to measure the performance of their current architecture. Several (application level) metrics are relevant to obtain insight in the impact of the workload on the performance:

Response time The response time of a request is an important indicator for the user perceived performance of the system. Webshop.com does not want the response time to increase compared to the monolithic application.

Number of requests per feature The number of requests per feature indicates how often a feature is used, and this can be compared to the number of requests of another feature. The number of products a customer adds to his shopping basket is a good metric to determine the impact of combining or splitting these features.

Data size Since duplication of features increases the storage requirements, it is important to know how much duplication occurs.

6 | Deployment Fitness

This chapter discusses various fitness objectives that are used to determine the quality of a deployment. The fitness of a deployment represents the quality of a deployment, given the defined objectives. At first candidate objectives originating from the field of performance modelling using queueing theory are discussed. Secondly candidate objectives originating from the field of software module clustering are described.

6.1 Queueing theory

Queueing networks are a well established method for performance modelling (Lavenberg, 1983). As indicated by the name, it is a network of queues, hence it is part of queueing theory, which is the mathematical study of queues (Allen, 2014) . Since computer systems can be represented as (networks of) queues and servers, this is a popular performance modelling technique.

The simplest queueing model is the so called M/M/1 queue. The first M, representing memoryless, means that the arrivals are determined by a Poisson process. The Poisson distribution is a discrete probability distribution for the counts of events that occur randomly in a given interval of time (Marchini, 2008). The second M means that the service time has an exponential distribution. Finally the 1 means that there is one server in the system. An example of a M/M/1 queue is depicted in figure 6.1.

Figure 6.1: Example of a M/M/1 queue

The λ shown in figure 6.1 is the symbol used to describe the mean arrival rate. The mean arrival rate is the mean rate at which new customers arrive at the system within a time unit. The mean processing rate μ describes the mean service rate of the server, i.e. how many customers the server can handle on average within a time unit. It should be noted that 'customer' is a common term in queueing theory used as a generic identifier of the input of the system. In this research, 'customer' can

be replaced with '(HTTP) request'. The mean service time can be derived from the service rate by calculating:

$$\text{mean service time} = \frac{1}{\mu} \tag{6.1}$$

Typically a first-in, first out (FIFO) queue is used, a queue in which the first customer added to the queue will be served first.

An important concept in queueing theory is the concept of utilization. Utilization describes the ratio of the mean arrival rate compared to the mean service rate. Utilization is thus defined as:

$$\rho = \frac{\lambda}{\mu} \tag{6.2}$$

The mean utilization of a system should be smaller than 1, since a utilization higher than 1 means that the mean arrival rate is greater than the mean service rate. This will result in an ever growing queue, since more customers arrive at the system than the system is able to process. As a result of an ever growing queue, the mean waiting time, the mean time a customer will spend in the queue, will grow to infinity. The mean waiting time can be calculated using the following formula:

$$W_q = \frac{\rho \cdot \frac{1}{\mu}}{1 - \rho} \tag{6.3}$$

It is easy to see that if ρ approaches 1, the mean waiting time will increase sharply.

Finally the mean sojourn time is an important concept in queueing theory. The mean sojourn time is the amount of time that a customer spends on average in the system. This means that the sojourn time is equal to the mean waiting time plus the mean service time. Definition 6.1.1 provides an overview of the definitions given so far.

Definition 6.1.1 (Overview of M/M/1 queueing theory definitions)
λ = Mean arrival rate
μ = Mean service rate
$\frac{1}{\mu}$ = Mean service duration
$\rho = \frac{\lambda}{\mu}$ (Utilization)
$W_q = \frac{\rho\frac{1}{\mu}}{1-\rho}$ (Mean waiting time)
$W_q + \frac{1}{\mu}$ = Mean sojourn time.

Based on the field of queueing theory, the following interesting objectives for the fitness function have been identified:

Mean sojourn time The sojourn time is in this case the time from which a change is confirmed by its public feature instance, till it has been propagated to all its internal feature instances in different microservices. Especially in an eventually consistent system, the mean sojourn time of requests should be as small as possible. The shorter the sojourn time, the less chance there is that a user views an inconsistent state of the system after making a change. Since the mean waiting time is a major component of the mean sojourn time, these objectives are both represented by this objective.

Utilization Common wisdom is to have utilization between 60 and 80 percent of the available capacity. By optimizing for microservice utilization in this range, the capacity of the cluster can be used more efficiently, while having sufficient capacity to handle peaks in the workload.

6.2 Software module clustering

In the related field of software module clustering, a genetic algorithm has also been used to solve the clustering of software modules based on the relationships among the modules (Mahdavi et al., 2003; Praditwong et al., 2011). They use several objectives to measure the fitness:

Maximize number of intra-edges Intra-edges are dependencies within a cluster. A high number of intra-edges indicates high cohesion.

Minimize number of inter-edges Inter-edges are dependencies between clusters. A low number of inter-edges results in low coupling.

Maximize cluster count To prevent a single huge cluster containing all modules.

Minimize single module clusters To prevent every module becoming its own cluster.

Since dependencies between microservices are resolved by adding an internal feature instance to a microservice, the number of internal feature instances reflects the number of inter-edges. Secondly it also determines how much data duplication occurs. In case a lot of data is processed by the application, it is beneficial to reduce the duplication of features and thus data, to reduce hardware costs. The other two objectives, maximizing cluster count and minimizing single module clusters, are not considered good objective in this case, since a deployment consisting of all features in individual microservices, or a single microservice might result in the best performance results according to the objectives discussed before.

6.2.1 Fitness objectives

To summarize, the following objectives have been discovered in field of queueing theory and software module clustering:

- Mean sojourn time
- Mean utilization
- Number of duplicated features

The mean sojourn time is directly related to the user perceived performance of a microservice system. As described before, this is an important factor in case of event-driven microservice architectures, that use asynchronous messaging between microservices to propagate changes. If the sojourn time becomes larger, it is easy to see that a user is more likely to see an inconsistent state of the system, by viewing data from an internal feature instance that has not processed the latest change yet. Hence it is also desired to keep the sojourn time as low as possible from a usability perspective.

Utilization is a measure of the used capacity. Unused capacity is basically wasted money for companies, hence they aim to maximally use the available capacity. Typically a utilization of around 60 till 80 percent is desired, since then capacity is efficiently used and it ensures there is capacity left to handle peaks in the load. By combining features together, the workload handled by a single microservice increases, which increases the utilization of such a microservice. However a utilization higher than 80 percent is also undesired, since that leaves too little capacity to handle unforeseen peaks in the workload. Hence this objective should be considered optimal when the utilization is between 60 and 80 percent. A too high or low utilization should be considered bad.

Finally the number of duplicated features is an indicator of how much data and code duplication occurs. The higher the number of duplicated features, the more asynchronous messaging between microservices has to occur, since a change needs to be propagated to more internal feature instances in different microservices. This might result in an increase of sojourn times, since microservices might have a lot of propagated messages in their queues. Finally duplication of features results in reduced maintainability and increased costs of storage of the data in case the microservice architectures handles large amounts of data.

7 | Improving the deployment

The previous chapter described the objectives to determine the quality of a microservice architecture. This chapter describes the algorithm used to search for better deployments given a workload. Secondly it describes the required operators to modify a deployment used by the optimization algorithm.

7.1 Optimization algorithm

This section examines which algorithm is suited for the problem at hand. The problem, the distribution of features over microservices, is closely related to the problem of software module clustering. Software Module Clustering is defined as automatically finding a good clustering of software modules based on the relationships among the modules (Mahdavi et al., 2003). In this field several optimization approaches have been proposed, such as hill climbing (Mancoridis et al., 1998, 1999) and genetic algorithms (Harman et al., 2002; Praditwong et al., 2011). Both methods use a fitness function to express the quality of the clustering. Since the approach using an genetic algorithm combined with a multi-objective approach by Praditwong et al. (2011) resulted in better results than hill climbing, it was decided to use a genetic algorithm to solve this problem. At the time this decision was made, it was still undecided whether a combined single objective approach or a multi-objective approach would be used, which made the genetic algorithm a better option.

7.2 Genetic algorithm

Genetic algorithms are a family of optimization algorithms inspired by Darwinian evolution (Holland, 1975; Whitley, 1994). A genetic algorithm is schematically depicted in Figure 7.1. The algorithm starts with an initial population of solutions. In genetic algorithm terms, every solution is called a chromosome. For each of these chromosomes, the fitness is evaluated. Next the termination condition is evaluated. In case the termination condition is met, the best chromosome from the population is returned and the algorithm stops. In case the termination condition is not met, the algorithm will select a subset of chromosomes from the population. These chromosomes will be used as input for the mutation and crossover phase. In this phase new chromosomes will be created by mutation and crossovers operators on the selected chromosomes. This will result in a new set of chromosomes, called offspring, that will

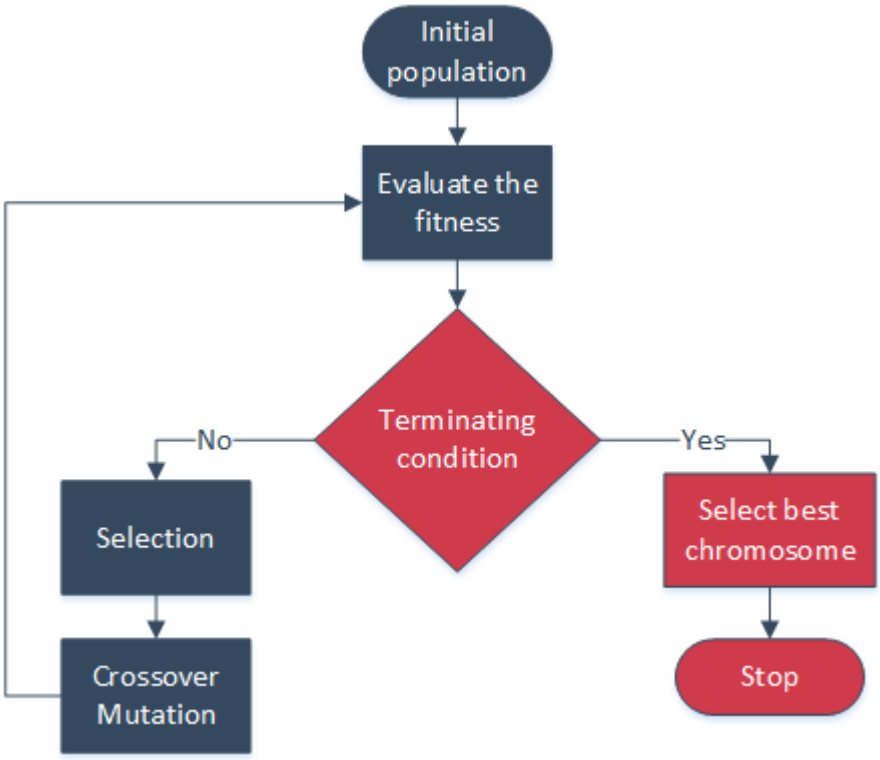


Figure 7.1: A schematic representation of a genetic algorithm

become the new population. This population will be the input for the next run of the algorithm.

In order to apply a genetic algorithm to solve a problem, the problem should be genetically encodable, such that the genetic operators mutation and crossover are able to transform a chromosome in a meaningful way. This genetic encoding, as the name states, is a representation of the problem that resembles the way DNA is represented. Typically this is depicted as an array of bits or characters, as shown in Figure 7.2. A bit or character in the genetic encoding, is called a gene.

Figure 7.2: Examples of genetic encoding

The selection phase in a genetic algorithm determines which chromosomes from the population are taken as input for the crossover phase of the algorithm. It is typical that the fitness of a chromosome is used in the selection process. This fitness describes the quality of a solution. The fitness of a solution is calculated by a using a problem specific fitness function. Several types of selection algorithms exist such as Elite, Roulette Wheel or Tournament selection. Elite selection orders the chromosomes by their fitness and selects a proportion of the chromosomes, which are reproduced 1/proportion times. Since Elite selection selects only the best candidates, it is likely to

get stuck in local optima. Tournament selection is a different selection algorithm that does not suffer from this problem. With tournament selection n random chromosomes from the population are selected. The chromosome with the best fitness will be selected for the crossover phase. Several tournaments are held to select multiple chromosomes till the desired crossover population size is obtained.

A crossover operator works on this genetic encoding. A crossover operator takes one or more chromosomes from the population as input, and creates one or more child chromosomes. This is typically done by combining the genes of the parent chromosomes, by for example switching all genes after a fixed crossover point, as shown in Figure 7.3.

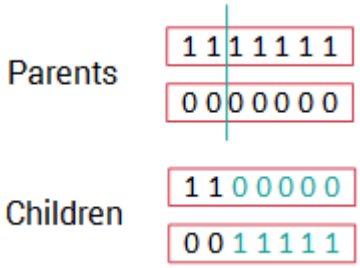


Figure 7.3: Example of single point crossover operator

If a genetic algorithm would only use a crossover operator, the algorithm would be sensitive to local optimal, since only chromosomes based on the original population would be examined by the algorithm. To solve this problem, the mutation operator is introduced. The goal of this operator is thus to increase the diversity of the chromosomes that are examined. A mutation operator operates on a single chromosome and it alters one or more genes of the chromosome. Since mutation applies (semi) random changes to a chromosome, the algorithm should visit more diverse solutions from the solution space. A typical implementation of a mutation operator creates a random number for every gene that determines whether this gene should be modified. Figure 7.4 shows an example of a mutation applied to a chromosome, that flips several bits of the chromosome. The probability of an occurrence of a mutation should not be set too high. If this value is set too high, it would result in the algorithm becoming a random search, which will result in an inefficient search for better deployments.

Figure 7.4: Example of mutation operator

7.3 Genetic problem encoding

In order to be able to use a genetic algorithm, the problem at hand first needs to encoded in a genetic representation. This section explains the encoding used in this research. A single chromosome in the population should represent a single deployment.

As described in section 4.1, a deployment can be described as a 4-tuple (M, I, λ, h) given a feature model $FSM = (F, P, R)$. Chromosomes should be encoded in such a way that it is possible to compare them.

The feature instances are thus suboptimal to represent a deployment, since the set of feature instances differs per deployment. A possible solution would be to include all feature instances in the encoding of a deployment. This would however result in overhead in the representation, since many feature instances are not present in a deployment.

However, the feature instances required for a deployment can be derived from the placement of the public feature instances over the different microservices combined with the relations. If the placement of public feature instances in different microservices is encoded in the genetic encoding, the dependency relations can be used to derive the internal feature instances that are required to fulfil the dependency relations for the deployment. Since the set of features of a feature model is stable across the deployments, this representation is an efficient encoding of the problem.

A simple representation for the placement of a public feature instance is to assign every microservice a unique incremental number as identifier. This microservice identifier is then used to identify in which microservice a public feature instance is located.

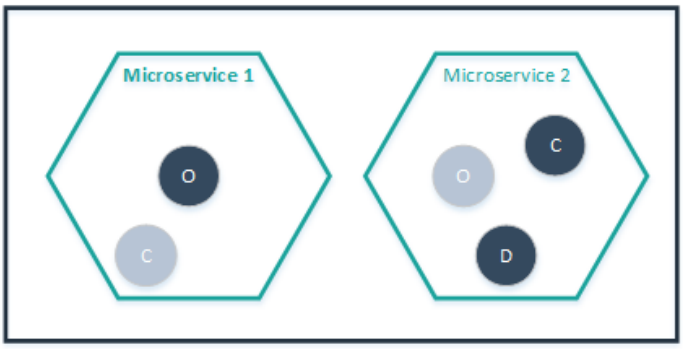


Figure 7.5: Deployment of the running example

An example of this encoding is applied to a deployment of the (partial) running example described in chapter 4. In this deployment the order feature is located in a microservice, and the customer and delivery feature are located in another, this is visualized in figure 7.5.

Table 7.1 shows a encoding of this deployment using the representation method discussed above. In this case, the order feature is represented as feature 1, the customer feature as feature 2 and the delivery feature as feature 3.

Table 7.1: Simple genetic encoding of the running example deployment

Feature	1	2	3
Microservice	1	2	2

However this encoding does not uniquely represent a deployment. The same deployment can also be represented by the encoding shown in table 7.2. Although the encoding is different, the same features are placed together in a microservice. Since a microservice is defined by the features that it contains, as defined in definition 4.1.2, this deployment is the same as the deployment described in table 7.1.

Table 7.2: Redundant genetic encoding of the same running example deployment

Feature	1	2	3
Microservice	2	1	1

A solution having more than one representative chain in the encoding scheme results in the encoding having redundancy (Menouar, 2010). The redundancy of this simple encoding is large, since a deployment with m microservices, can be represented by $m!$ different chains. Since the redundancy grows exponentially for the number of microservices, a large part of the domain of the genetic encoding consists of duplicate chains.

To solve this problem, the microservice identifier should be deterministically derived in such a way that the same features in a microservice result in the same microservice identifier. Hence the identifier should be based on the features contained in the microservice. Instead of assigning an incremental integer as identifier of the cluster, the highest feature number of the features located in the microservice is chosen as microservice identifier. An example of this encoding applied to the running example deployment is shown in table 7.3. Since feature 1 is the only feature in its microservice, it also has the highest feature number, hence this microservice identifier becomes 1. For the second microservice, there are two features: feature 2 and 3. Since feature 3 has the highest feature number, the microservice identifier becomes 3. It should be noted that the microservice identifiers in this encoding are no longer incremental. Secondly we have chosen the highest feature number arbitrarily, the lowest feature number works equally well.

Table 7.3: Non redundant genetic encoding of the running example

Feature	1	2	3
Microservice	1	3	3

7.4 Genetic operators

As described earlier in this chapter, both the crossover operator and the mutation operator create new chromosomes by modifying existing chromosomes. This section describes these operators and their relation with the microservice model described in chapter 4.

7.4.1 Crossover operator

The genetic encoding presented in the previous section is a position based encoding, i.e. a particular feature must always be located at the same position in the encoding. A crossover operator should create one or more new deployments, called child chromosomes, based on one or more chromosomes of the current population, called parent chromosomes. We decided to create a crossover operator that creates two child chromosomes by merging two random microservices of two parent chromosomes. Combined with an initial population of chromosomes in which every feature is located in its own microservice, this crossover operator combined with the mutation operator resulted in an efficient search through the search space.

This crossover operator works by first selecting a random microservice in the first parent chromosome. This same procedure is repeated for the second parent chromosome. Next all features of the first chromosome are selected that are located in the selected microservice. The same procedure is again repeated for the second parent chromosome. Afterwards two child chromosomes are created, one cloned from the first chromosome and another cloned from the second parent chromosome. For both child chromosomes all selected features are placed together in a single microservice.

Microservice architecture model

The crossover operator essentially performs a merge of two microservices. This section defines the *merge* function on the microservice architecture model defined in chapter 4.

At first several primitives operators and functions on features instances I are defined, since they are required for the more complex *merge* operation. A helper function s is defined that indicates if two feature instances are of the same feature:

Definition 7.4.1
Given two feature instances i and j, the function s, that defines if two feature instances are of the same feature, is defined as:

$$s(i,j) = \exists f \in F : \lambda(j) \subseteq f \wedge \lambda(i) \subseteq f$$

The lowest level operator is the merge of two feature instances that are instances of the same feature. The merge of an internal feature instance with its public feature instance is defined as the public feature instance, since the internal feature instance must represent a subset of the properties of its public feature instance, i.e. $\lambda(i_{internal}) \subseteq \lambda(i_{public})$. The result of merging of two internal feature instances is defined as an internal feature instance that has the same properties as the union of the properties of the two feature instances. The merge of two feature instances is denoted by $\oplus$, as shown in definition 7.4.2.

Definition 7.4.2 ($\oplus$ operator for feature instances)
Given two sound feature instances i_j, i_k such that $s(i_j, i_k)$, the $\oplus$ operator is defined as:

$$i_j \oplus i_k = \begin{cases} i_j & \text{if } i_j = h(\lambda(i_j)) \\ i_l \text{ such that: } \exists i_l \in I : \lambda(i_l) = \lambda(i_j) \cup \lambda(i_k) & \textit{otherwise} \end{cases}$$

Applying the $\oplus$ operator to two sound feature instances should result in a sound feature instance. The only constraint defined on a feature instance is that it represents a subset of the properties of its feature. In case the $\oplus$ operator is applied to a public feature instance and a internal feature instance, the resulting feature instance is the public feature instance. Since this public feature instance is a sound feature instance by definition, the first case of the definition is unable to break the constraint. In the other case, two internal feature instances are merged, resulting in a feature instance that represents the same properties as the union of the properties of both feature instances. Since both feature instances contain a subset of the properties of the feature they represent, it is easy to see that the union of these properties cannot be a strict superset of the feature they represent. Hence $\oplus$ always returns a sound feature instance when applied to two sound feature instances.

Based on the $\oplus$ operator, it is possible to merge microservices. A merge of two sound microservices is defined as a microservice of which the feature instances represent the same properties as the union of the properties represented by the individual microservices and that contains all public feature instances present in both microservices. In case a feature is represented by a feature instance in both microservices, these feature instances should be merged, in order to fulfil the constraint that every microservice only contains each feature once. In case a feature is only represented in one of the microservice, its feature instance can be added to the resulting microservice. The *merge* is defined in definition 7.4.3.

Definition 7.4.3 (Merge function for microservices)
Given two microservices m_i, m_j, the merge operator $\uplus$ is defined as:

$$\begin{aligned} m_i \uplus m_j = &\{i \oplus j \mid i \in m_i, j \in m_j : s(i,j)\} \\ &\cup \{i \mid i \in m_i, \forall j \in m_j : \neg s(i,j)\} \\ &\cup \{j \mid j \in m_j, \forall i \in m_i : \neg s(i,j)\} \end{aligned}$$

The resulting microservice of a merge of two sound microservices should be sound as well. The first constraint that a microservice should contain all feature instances to fulfil its dependency requirements cannot be broken by *merge*. Both m_i and m_j are sound microservices and should thus contain all feature instances needed to fulfil the dependency requirements. In case a feature is located in both individual microservices, the merged feature instance of these feature instances is placed in the merged

microservice. As a result of the definition of $\oplus$, all properties represented by both feature instances are represented by the merged feature instance, hence the dependencies are still fulfilled. In case a feature is only represented by a feature instance in one of the microservices, its feature instance is placed in the merged microservice. Since both microservices are sound, this feature instance already fulfilled the dependency requirements of its dependants and it will as well fulfil these in the merged microservice.

7.4.2 Mutation operator

Since the crossover operator converges rather quickly, the goal of the mutation operator is too widen the search area. This is done by randomly selecting a gene from the deployment. Next all features located in the same microservice are moved to a random microservice. This might either be an existing microservice, or a new one. While this is a mutation with a large impact, it resulted in better coverage of the search space compared to less destructive mutations.

Microservice architecture model

Since the mutation operator moves features, the *move* function has to be defined on the microservice architecture model.

A move of a feature is essentially removing the feature in its original microservice followed by a insert in its new microservice. Hence both *remove* and *insert* will be defined as sub-functions. At first the *include* function, that adds a feature instance without dependencies to a microservice, is defined

Definition 7.4.4 (Include function for feature instances)
Given a sound microservice m and a sound feature instance without dependencies i, the *include* function is defined as:

$$include(m, i) = \begin{cases} m \cup \{i\} & \text{if } \nexists j \in m : s(i, j) \\ \{i_m | i_m : \neg s(i, i_m)\} \cup \{i \oplus i_m | i_m : s(i, i_m)\} & otherwise \end{cases}$$

The *include* of a feature instance without dependencies i has an impact on the constraints defined on a microservice and a deployment. The first constraint is that every microservice contains all instances necessary to fulfil the dependency requirements. Since i is defined as a feature instance without dependencies, m still fulfils all dependency requirements after i being included. The second constraint is that every feature has at most one feature instance in a microservice. If m does not contain a feature instance of the same feature as i, it is easy to see that this constraint still holds after adding i to m. In case m does already contains a feature instance of the same feature as i, the second case of the *include* function is applicable, and the new microservice is formed by all feature instances of m that are instances of

a different feature and i merged with the feature instance that represents the same feature in m. Again it is easy to see that the feature of i is only present once in the resulting microservice. By definition the included feature instance i is a subset of the feature, and thus the constraint that every feature instance is a subset of its feature still holds. Given a sound deployment, the addition of a feature instance is unable to violate the constraint that every microservice contains at least one public feature instance. However, it is possible to violate the constraints of M being a partitioning of I, by adding a feature instance i to m that is already present in another microservice. Based on *include*, the *insert* function, that correctly inserts feature instances with dependencies, can be defined as shown in definition 7.4.5:

Definition 7.4.5 (Insert function for feature instances)
Given a feature model (F, P, R), a sound microservice m and a sound feature instance i with dependencies, the *insert* function is defined as:

$$insert(m, i) = include(m, i) \uplus \biguplus_{\{j|\lambda(j)=\{q\} \text{ where } (p,q)\in R, p\in\lambda(i), i\in m, j\notin m\}} insert(m, j)$$

Since *insert* uses *include*, it can break the same constraints as include, namely that M is no longer a partitioning of I. The other constraints are discussed below:

Every microservice should contain all feature instances to fulfil its dependency requirements This constraint is met by the *insert* function. In case the inserted feature instance has no dependencies, than the *include* is called, which is shown above to adhere to the constraint. In case the inserted feature instance has dependencies, the second case is applicable, which includes i itself using the *include* function, and then merges the resulting microservice with all microservices that are created by adding a dependency of i to m. Since the function recurses on the added dependencies, all recursive dependencies are added as well. The function is capable of handling cyclic dependency graphs, since the constraint $j \notin m$ ensures that every feature instance is only inserted once.

A microservice has at most one feature instance of a feature In the first part of the function an *include* of i occurs, which does not violate this constraint. In the second part two microservices are merged. Given the definition of merge above, it is easy to see that merge is unable to violate this constraint as well. As a result, *insert* is unable to break this constraint.

A feature instance should be a subset of its feature By definition i is a subset of its feature and R contains only properties of P. This makes it impossible to violate the constraint that every feature instance is a subset of its feature.

In order to complete the *move* function, the *remove* function is required. The goal of this function is to remove a feature instance from a microservice, as shown in 7.4.6.

Definition 7.4.6 (Remove function for feature instances)
Given a sound microservice m and a sound feature instance i, *remove* is defined as:

$$remove(m, i) = \{j | j \in m : j \neq i\}$$

Since the *remove* function removes a feature instance from a microservice, it is easy to see that it might remove a dependency of another feature instance. Hence the constraint that microservice m contains all feature instances to fulfil the dependency requirements can be violated by *remove*. The second constraint, that a microservice has at most one feature instance of every feature, can not be violated by remove, given m is a sound microservice. If m is sound, it should contain at most one feature instance of every feature. After removing a feature instance, it is easy to see that for every feature there is still at most one feature instance. Given that m is a sound microservice, *remove* is unable to violate the constraint that every feature instance is a subset of its feature. However, *remove* is able to break the constraint that each feature has a public feature instance in a deployment, as well as the constraint that every microservice has at least one public feature instance.

Since the removal of a feature instance in a as part of the to be defined *move* function might result in unmet dependencies, a function is needed that reinserts all missing dependencies as internal feature instances. The second part of *insert* function reinserts all missing dependencies, so the microservice becomes sound again. This part is extracted as a separate *repair* function:

Definition 7.4.7 (Repair function for microservices)
Given a feature model (F, P, R), a microservice m in a sound deployment d, the *repair* function is defined as:

$$repair(m) = \biguplus_{\{j|\lambda(j)=\{q\} \text{ where } (p,q)\in R, p\in\lambda(i), i\in m', j\notin m'\}} insert(m', j)$$

where

- $m' = \{i | i \in m \wedge h(\lambda(i)) = i\}$

The *repair* function works by first selecting all public feature instances in the microservice. All dependencies of these public feature instances are then added by calling *insert* for every dependency. As shown above, *insert* recursively adds all dependencies. As a result, *repair* returns a microservice that contains all feature instances needed to fulfil the dependency requirements. The definition of *insert* ensures that this function is unable to insert a feature instance of the same feature in a microservice, but instead will merge both feature instances of the same feature. Since j represents a single property q of P, j will always be a subset of a feature,

hence each resulting feature is a subset of its feature. Given a microservice m in a sound deployment d, the repair function will not remove any public feature instances, hence every feature will still have its public feature instance. Using *repair* on a microservice without public feature instances will result in $\emptyset$, hence it is able to create a deployment with a microservice that has no public feature instances.

Now all components are defined, the *move* function can be defined, that moves a microservices from a microservice to another microservice:

Definition 7.4.8 (Move function for feature instances)
Given a deployment (M, I, λ, h), two microservices m_{from}, m_{to} in M and a public feature instance i located in m_{from} , the *move* function is defined as:

$move(M, m_{from}, m_{to}, i) = \{f(m) | m \in M, f(m) \neq \emptyset\}$
where

- $f(m) = \begin{cases} repair(remove(m_{from}, i)) & \text{if } m = m_{from} \\ insert(m_{to}, i) & \text{if } m = m_{to} \\ m & \text{otherwise} \end{cases}$

While most of the previous functions were able to violate one or more constraints of a sound deployment, the *move* function is unable to produce an invalid deployment when applied to microservices of a sound deployment. The *remove* function is able to remove dependencies of other feature instances, but any missing dependencies are restored by the *repair* function. Secondly it is possible to break the constraint that every feature has a public feature instance in the deployment, by removing it. However since the same public feature instance is added to m_{to}, the *move* function is unable to violate this constraint. In case m_{to} does already contain a feature instance of the same feature, *insert* merges i with this feature instance. Hence *move* is unable to break the constraint that a microservice contains at most one feature instance of a feature. Since *repair* returns $\emptyset$ in case a microservice no longer has a public feature instance, and empty sets are not added to the resulting partitioning, *move* is unable to break this constraint as well.

7.5 Fitness calculation

The fitness function determines how fit a deployment is, i.e. the quality of the deployment, given the defined goals. Typically the goal of the genetic algorithm is to maximize the fitness score. The fitness function is calculated for every deployment evaluated by the genetic algorithm, hence its speed is crucial for the speed of the genetic algorithm. In some cases calculating an exact fitness might not be computationally feasible. In these cases an approximation of the fitness can be used.

This section describes how the fitness can be calculated for the deployments created by the crossover and mutation operators. Of course the fitness can be calculated

by actually implementing the deployment and executing the workload on the deployment. Based on the performance metrics, the fitness with regards to the defined objectives can be determined. However evaluation of a deployment becomes very cumbersome and time consuming. Hence approximations of the performance of a deployment are required.

7.5.1 Queueing theory

Queueing theory is a good candidate for approximation of the performance of a deployment. Although the M/M/1 model discussed earlier is the most simple queueing model, it has been proven useful in several real life scenarios. It has been in used for example in the performance analysis of cluster based web services (Levy et al., 2003) and for multi-tier internet services (Urgaonkar et al., 2005). However, it is not expressive enough to model a microservice deployment correctly.

By extending the M/M/1 model by introducing customer classes, the model becomes expressive enough. An example of a M/M/1 queue with multiple customer classes is depicted in figure 7.6.

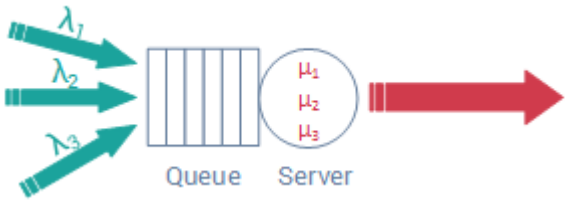


Figure 7.6: Example of a M/M/1 queue with multiple customer classes

The standard M/M/1 model assumes that all requests are of the same type, and have an exponential distribution. A microservice containing several features, will process requests of different types. Based on the type of requests, it is very likely that different types of requests have a different exponential distribution. As in a M/M/1 model, the arrival rate is a Poisson process which can be merged and split (Lavenberg, 1983) based on the chance that a request has a certain class.

Thus, each class has its own service rate, denoted by μ_i, arrival rate, denoted by λ_i, and a chance of occurring, denoted by p_i. In this model, the total arrival rate of the server is:

$$\lambda = \sum_{i=1}^{n}(\ p_i \cdot \lambda_i \ ) \tag{7.1}$$

, i.e. the mean of the mean arrival rates of the different customer classes. Similarly, the service time is calculated as the weighted sum of the service times of the different classes. Hence, the mean service time of the server can be calculated using the following formula:

$$\mu = \sum_{i=1}^{n}(\ p_i \cdot \lambda_i \cdot \mu_i \ ) \tag{7.2}$$

Furthermore the mean waiting time at the server can be calculated using the same formula as in the case of a simple M/M/1 model.

As an approximation for the chance that a request is of a certain class, we assume the classes to be uniformly distributed, i.e.,

$$p_i = \frac{\lambda_i}{\sum_{i=1}^{n} \lambda_i} \tag{7.3}$$

While it is possible to retrieve the mean processing times and arrival rates per microservice for the current deployment based on the performance metrics of the running system, this is not possible for the new deployments generated by the mutation and crossover operators. Since it takes too much time to generate the load on an actual implementation of the suggested deployment, it should be possible to derive the new arrival rates, processing times and resulting waiting times based on the deployment and the performance metrics of the current deployment.

It is important to obtain the performance metrics for individual features, since splitting performance metrics is not supported by the framework. For both modification operators, *merge* and *move*, it needs to be determined how to calculate the performance of the resulting microservice.

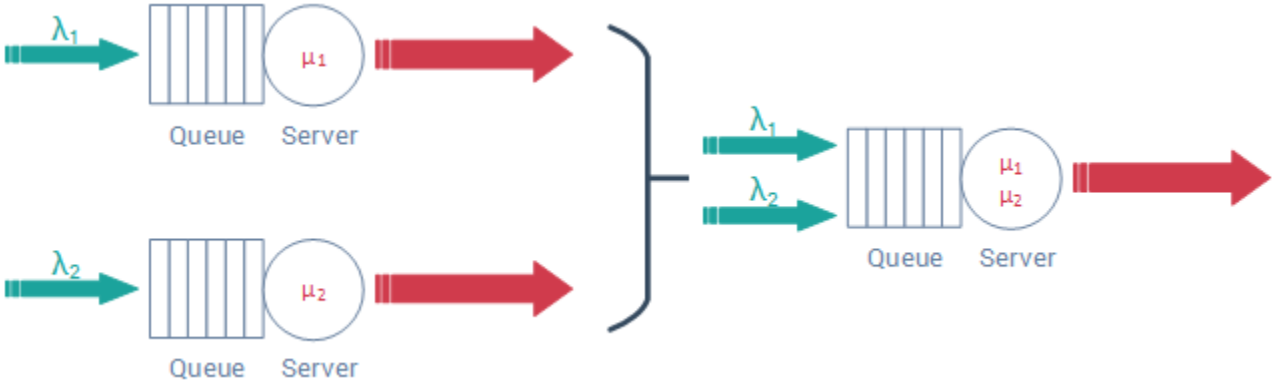


Figure 7.7: Example of merging two M/M/1 queues

In case a merge of two microservices is performed, the new arrival rate, service rate and waiting time needs to be derived. A visual representation of such a merge is shown in figure 7.7. For simplicity the following notation is introduced:

$$m_i = (\lambda_i, \mu_i) \tag{7.4}$$

This notation represents a microservice i that has a mean arrival rate λ_i and a mean service rate μ_i. Now a merge of two microservices m_i and m_j with different customer types can be defined as:

$$m_i \oplus m_j = (\lambda_{merge}, \mu_{merge}) \tag{7.5}$$

It is easy to see that λ_{merge} is simply the sum of the arrival rates of the individual microservices λ_i and λ_j, since the resulting microservice has to service customers of both types.

$$\lambda_{merge} = \lambda_i + \lambda_j \tag{7.6}$$

This calculation can not be applied to the service rate, since if the service rates would be added, the resulting microservice would be able to process at the rate of both individual microservices combined, which is not realistic. Hence the mean of both μ_i and μ_j seems more appropriate, i.e. the mean service rate of the merged microservice is the mean of the service rates of both individual microservices. Although this seems logical, it has to be improved, since the mean assumes that both type of customers occur with 50% chance, however this assumption does not hold for all cases. It is easy to think of a scenario in which λ_i is much larger than λ_j. μ_{merge} would be skewed towards μ_j with a simple mean. Hence a weighted mean based on the arrival rate of the different customer types seems more appropriate. This results in the following formula:

$$\mu_{merge} = \frac{\lambda_A}{\lambda_A + \lambda_B} \cdot \mu_A + \frac{\lambda_B}{\lambda_A + \lambda_B} \cdot \mu_B \tag{7.7}$$

Based on the calculated mean arrival rate and the mean service rate, the waiting time can be calculated using the same formula for simple M/M/1 queues.

Definition 7.5.1 (Merging two microservices)
Given two microservices $m_A = (\lambda_A, \mu_A)$ and $m_B = (\lambda_B, \mu_B)$, their merge, denoted by $m_A \oplus m_B$ is again a microservice, defined by

$$m_A \oplus m_B = (\lambda_A + \lambda_B, \frac{\lambda_A}{\lambda_A + \lambda_B} \cdot \mu_A + \frac{\lambda_B}{\lambda_A + \lambda_B} \cdot \mu_B)$$

The previous paragraph described what happens when two microservices are merged that serve different customer classes, i.e. have different features. However merging microservices that process the same customer class, requires different formulas. In the previous formulas, the arrival rate of a shared customer class would be added twice, while it should be added only once. Secondly the mean service rate would be skewed towards the mean service rate of the shared customer class, since that would be used twice in the calculation of the weighted mean. It is easy to see that the correct mean arrival rate can be calculated by counting the shared customer class only once. The mean service rate is more difficult, since both microservices have different service rates for the same customer type. It is hard to determine in general what the effect is of a merge on the service rate. Depending on the type of insights desired, it is suggested to take the minimum, mean or maximum of the service rates. In case the minimum of both service rates is used, the most pessimistic analysis is performed. This might be useful if worst-case scenario insights are desired. The mean of both service rates will result in a mean service rate that in some cases might be a bit too fast, and sometimes too slow. Hence in general the mean should result in

reasonably good results. In case you are unsure which to use, the mean is suggested. Finally the maximum results in the optimistic view on the system. Once again the calculation of the utilization and the waiting time does not change.

The formulas described in this section can be used to analyse the performance of a deployment created by crossovers and mutations based on the initial deployment and the corresponding metrics.

8 | Case study

A case study was performed to evaluate the approach. This chapter describes the case study context and the results.

8.1 Case study context

This research is performed at AFAS Software B.V. The vision of AFAS is to automate all (administrative) business processes. The main product of AFAS is an Enterprise Resource Planning (ERP) system. The ERP system is offered as a Software-as-a-Service (SaaS), called AFAS Online, or it can be hosted by the client on premise. Since 2011 AFAS also offers an online expenditure book aimed at consumers, called AFAS Personal.

In 2016, AFAS Software had a total revenue of more than 100 million euros and a profit of 36 million euros. They employ 382 employees, of which 363 full-time employees. Together these employees serve over 1 million users worldwide. Their headquarters is located in Leusden in the Netherlands. Besides their office in Leusden, they have offices in Belgium, Curaçao and Aruba.

AFAS was founded in 1996 as AFAS Automatisering, as a result of a management buyout. Shortly after the buyout, the company was renamed to AFAS Software B.V. In 1999, AFAS Personele Systemen was founded, focusing on HR and Payroll software. Both companies merged in 2002, resulting in AFAS Software.

8.1.1 Profit Next

Profit Next is the next version of their ERP product. The vision for Profit Next is that programming should be automated, with a separation between functionality and technique. The NEXT version of AFAS' ERP software is completely generated, cloud-based and tailored for a particular enterprise, based on an ontological model of that enterprise. The ontological enterprise model will be expressive enough to fully describe the real-world enterprise of virtually any customer, and as well form the main foundation for generating an entire software suite on a cloud infrastructure platform of choice: AFAS NEXT is entirely platform- and database-independent. AFAS NEXT will enable rapid model-driven application development and will drastically increase customization flexibility for AFAS' partners and customers, based on a software generation platform that is future proof for any upcoming technologies. Currently the generated architecture is an event driven microservice architecture using Command

and Query Responsibility Segregation (CQRS) (Kabbedijk et al., 2012), and Event Sourcing (Overeem et al., 2017) running on Microsoft Service Fabric at the back-end, and a HTML5 single page application as front-end.

Profit Next's generator supports the generation of different feature groupings at the query side of its CQRS backend. The generator was modified to support a deployment model, as described in section 4.3, as auxiliary input for the generation process. This deployment model is used to distribute the query handlers over the different microservices. Because of the ability to modify the application generation process, it is possible to generate many different groupings of features. This makes Profit Next an excellent case study environment for this research.

8.1.2 Microservice Architecture Model

The NEXT platform was used to create an application that resembles a web-shop, depicted in figure 8.1. It contains a *customer* model element that enables visitors to create an account. A customer represents a person and consists of an email, password, default shipping address, and other personal details. Secondly the web-shop contains *products*, consisting of product information and several technical properties. These products can be reviewed by a customer, using the *review* event. A customer can create *orders*, consisting of one or more *order lines* containing an amount and a product. Furthermore an order consists of a shipping address, by default the customers's default shipping address. Finally an order results in a *payment* and a *delivery*. A delivery is an event that results in goods leaving the organisation, requiring a payment in return of the other party, as depicted by the properties of delivery in figure 8.1. A payment contains a dependency on the total price of an order. The delivery is performed by a *parcel service*, and uses the shipping address provided on the order.

These six model elements result in 27 features with a total of 238 properties and 72 dependency relations between features. The maximal microservice architecture, i.e. every feature in its own microservice if possible, is used as a baseline for the performance tests. The resulting microservice architecture consists of 25 microservices, 27 public feature instances and 55 internal feature instances, as shown in figure 8.4(a). The number inside a feature indicates of which model element it originates. Features with a 1 or 2 originate from the sales order model element. Features 3 to 6 are created as result of the customer model element, and features 7 till 10 are the result of the Delivery element. The parcel service model element resulted in features 11 to 13. The review event resulted in feature 14, while the product role resulted in 15 and 16. Finally NEXT by default generates features 17 till 27, that were not used in this workload.

8.1.3 Workload

We created an artificial workload, since AFAS NEXT is still under development and not running in production. The workload we created consists of a typical scenario for a web shop. At first a user creates a shopping basket and adds several products to

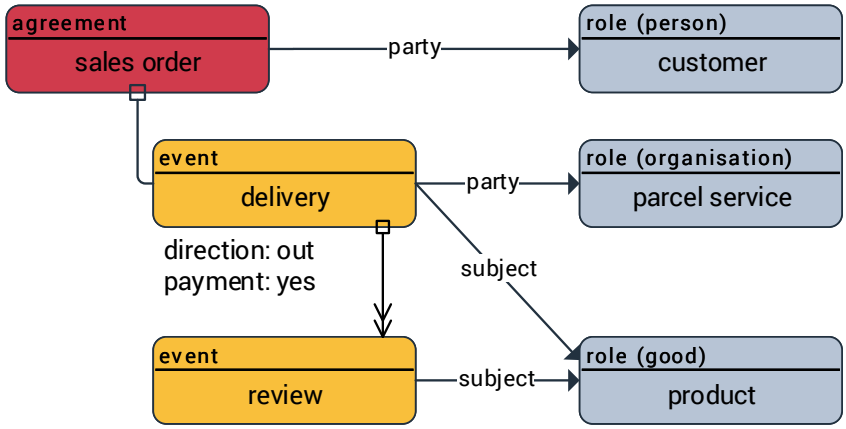


Figure 8.1: Model of the created AFAS NEXT application

it. When the customer is done shopping, he pays for his order, and a delivery slip is created. Afterwards, some users submit a review of the product.

8.1.4 Test setup

All tests are performed on a performance testing cluster. The complete testing setup is depicted in figure 8.2. This cluster consists of 5 virtual Windows Server 2012 machines with four cores and 10 GB RAM. Furthermore there is a quad core Windows Server 2012 virtual machine for the database server, running PostgreSQL 9.5, having 15 GB of RAM. The tooling created by Guelen (2015); Maddodi et al. (2016) is used to generate load based on the specified workload on the application.

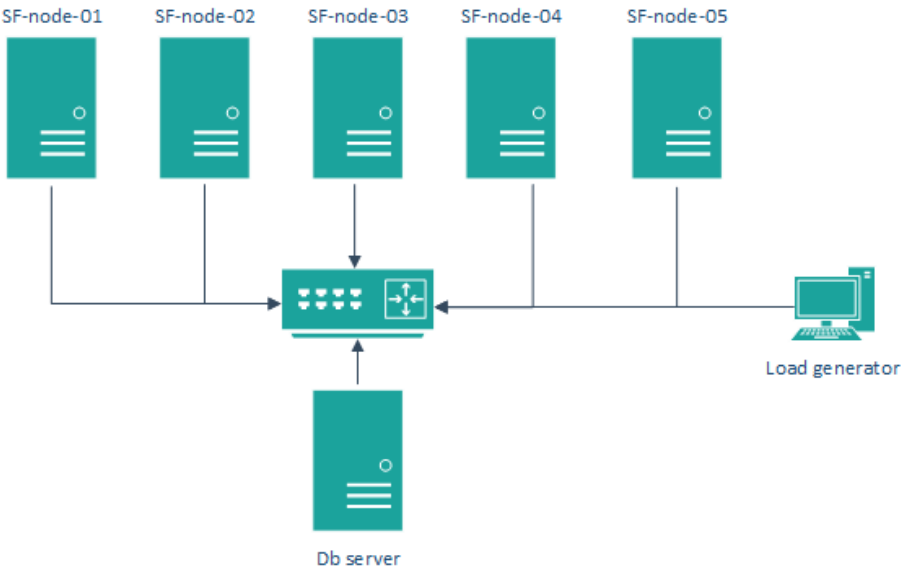


Figure 8.2: Case study test setup

The tooling by Guelen Guelen (2015) is used to generate the workload. Unfortunately the tooling is currently not able to send a single request and wait till the event has been processed by all projectors on the query side. Since the steps in the workload require the system to be consistent after every step, this scenario had to be converted to a batch workload to circumvent this limitation. This resulted in a phase for every step described in the workload above, in which all concurrent users perform that step. For example in the first phase all shopping carts are created, followed by a second phase in which all products are added to all shopping carts.

Fitness function

Since this workload is a burst process, which is not a poisson distribution, we were unable to use our queueing theory approximation and created a simulation that simulates the architecture of a Profit Next application. The output format of the simulation is identical to the queueing theory fitness function, hence it could be easily plugged in the genetic algorithm framework.

The simulation uses the workload as input and uses the average processing time of all events of a certain customer class as average service time. Based on these average service times, a discrete event simulation is created that simulates the processing of all requests in the system. The resulting waiting time, utilization and sojourn time of each event are calculated as part of the simulation. After the simulation has completed, the aggregations of these metrics are calculated and used as input of the fitness function.

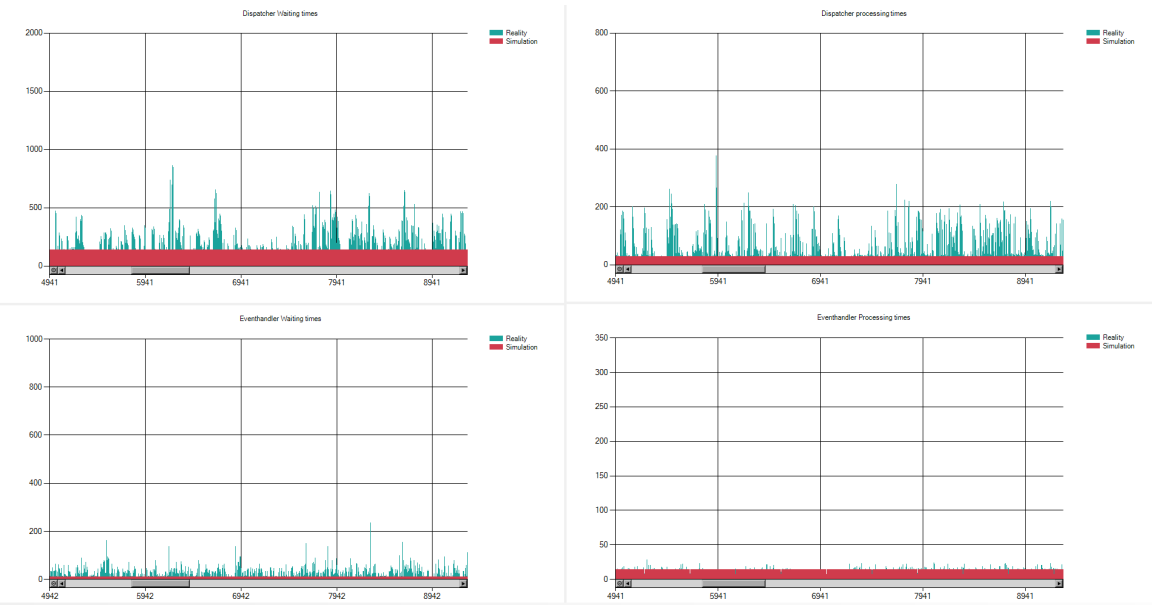


Figure 8.3: Simulation compared to reality

As described before, the simulation uses average service times, resulting in less variation compared to an actual run. However, it follows the same trends as an actual run, as shown in figure 8.3. Figure 8.3 shows a part of the entire overview, since the charting library produces an incorrect chart for the entire overview. The graph in the right upper corner of figure 8.3 displays the dispatcher service time. The dispatcher is an intermediate component that distributes a single event over all event handlers (microservices). The left upper corner depicts the waiting time at a dispatcher. The bottom right graph displays the processing times of the event handlers. Finally the left bottom graph shows the waiting times occurring at the event handlers. If the actual service times were used as input of the simulation instead of the global averages, the simulation was able to quite accurately simulate reality. However some parts of reality are not modelled in the simulation, such as Service Fabric rebalancing the cluster. If Service Fabric detects that the workload is unevenly spread over the cluster, it moves microservices to other nodes on the cluster. During this rebalancing, the balanced service temporarily has reduced throughput, resulting in increased event waiting times. It is easy to see that this has impact on the performance of the application, however it is outside of the scope of the simulation.

Fitness objectives

To evaluate our tool, we created an AFAS specific metric input module that derives both the workload and the performance metrics from their Software Operation Data. Based on goals of AFAS, we decided to use two fitness objectives: mean time till consistency with a weight of one, and feature duplication with a weight of 0.2. Both objectives had to be minimized. Furthermore the simulation was integrated in the fitness function.

Test variants

Before each performance test, base data is inserted in the system, such as accounts and products. In this phase, 251 countries, 250 users, 1760 products and four parcel services are created, that are used in the other phases of the performance test.

Three variants of the workload described in the previous subsection were used: the low, medium, and high variant. The low variant simulates traffic representing only a few users. This is done by running a single load generator thread that waits uniformly between 50 and 200 milliseconds between each request, i.e. the application has to handle between 5 and 20 requests per second. This tests is designed to determine a good clustering of features for small customers. Since the load on the system is low, the system should be immediately consistent, and no significant queueing of requests should occur.

The medium variant uses four load generators with the same settings as in the low variant case. The goal of this scenario is to create queueing at several microservices that handle a lot of updates. The system might need some catch-up time to become consistent, but should be quite quickly.

The high variant simulates a busy day for the web-shop. This is done by using ten load generators with the same settings as in the low variant. As a result, the application has to handle between 50 and 200 requests per second. The workload should result in significant queueing occurring at most places in the application. As a result of this queueing, the system will require several minutes catch-up time to become consistent.

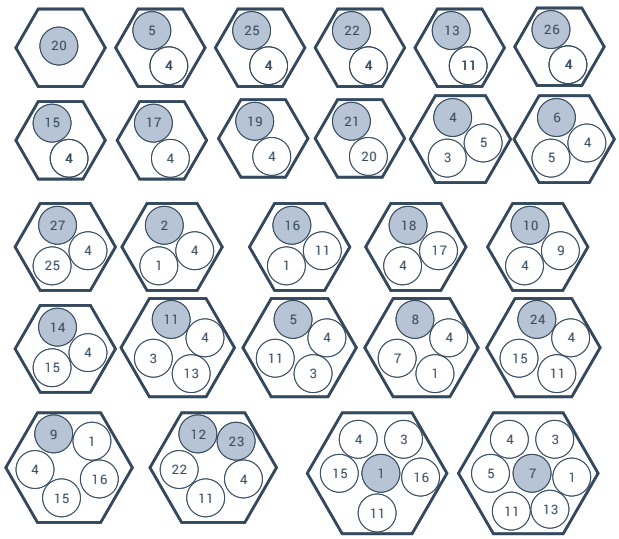
For all variants, five runs of the performance tests on the maximal microservice architecture depicted in figure 8.4(a) were performed. Afterwards the run that is the closest to the mean of the five runs was selected. This run was used as input for our tool, since the metric input module operates directly on the metrics. The microservice architecture model suggested by our tool was used as input for the AFAS NEXT generator. The regenerated application was redeployed, and the performance tests were re-ran on this new architecture. The following section discuss the results for the low, medium, and high workload scenarios.

8.1.5 Results

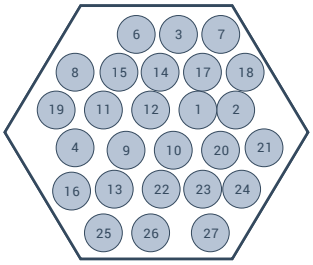
Low Workload Scenario

The performance test results of the maximal deployment for this scenario are shown in the left part of table 8.1. Every row in this table represents one of the phases of the workload. The time column denotes the total time in seconds it took the system handle this workload and become consistent. The avg. requests column denotes the mean number of requests the system handled per second. Note that a lower time is better, while a high number of avg. requests per seconds is better.

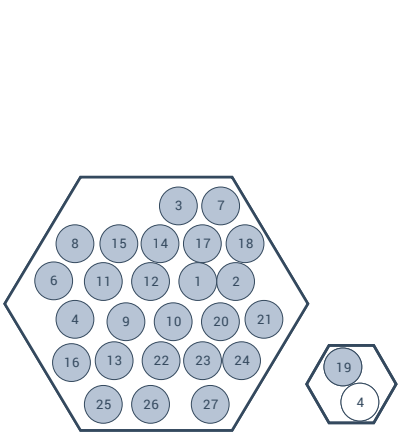
The metrics emitted during this test run were used as input for our tool. It recommends to merge all features in a single microservice, resulting in a microservice containing 27 public feature instances with zero duplication, as shown in figure 8.4(b). According to the simulation this should reduce the mean time till consistency from



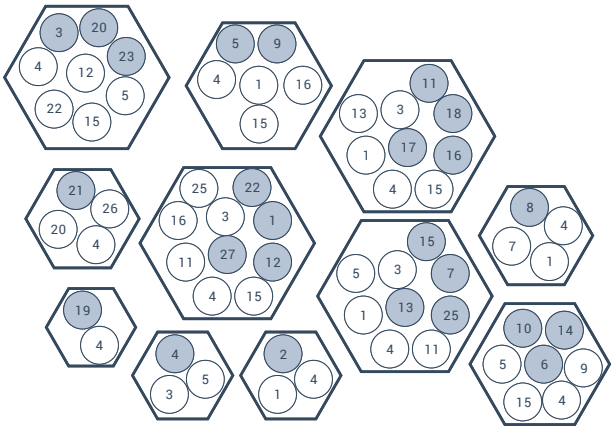
(a) Initial microservice architecture of the application



(b) Recommended architecture of the application for the low traffic workload



(c) Recommended architecture of the application for the medium traffic workload



(d) Recommended architecture of the application for the high traffic workload

Figure 8.4: Different microservice architectures for the feature model depicted in figure 8.1

183 to 62 milliseconds, while reducing the number of internal features instances from 55 to zero.

The results of the performance tests on the deployment based on the suggestion of our tool are shown in the right part of table 8.1. It should be noted that all phases of the performance test were completed faster compared to the the initial microservice architecture. Secondly the mean number of requests per seconds is higher for every phase.

Table 8.1: Performance test results for the low workload variant, before and after optimization with our tool and redeployment of the Microservice Architecture.

Test phase	**Initial MSA**		**Optimized MSA**	
(# requests)	Time (s)	Avg. requests/s	Time (s)	Avg. requests/s
Order (600)	131	4.56	104	5.75
Article (1200)	270	4.44	208	5.74
Payment (600)	132	4.53	104	5.53
Delivery (600)	126	4.74	108	5.53
Review (500)	108	4.61	89	5.58

Medium Workload Scenario

The results of the medium workload scenario performance test on the maximal deployment are shown in the left part of table 8.2. This table has the same layout as table 8.1, the table denoting the results for the low workload scenario.

Based on the metrics that were emitted during the test, our tool suggests the deployment shown in figure 8.4(c). The mean waiting time should be reduced from 154 to 71 according to the simulation, with only a single duplicated feature. The notification component is placed in a separate microservice. This feature provides users with notifications of comments among others. According to the simulation, moving this feature to its own microservice results in a reduction of the average sojourn time of 3 milliseconds compared to having all features in a single microservice.

The results of the performance test on the deployment suggested by our tool are shown in the right part of table 8.2. It should be noted that again all phases of the performance test completed in less time than in base case on the maximal deployment.

Furthermore it is interesting that the application processed less requests per second on average, while the system has processed the entire workload earlier, as shown in table 8.2.

High Workload Scenario

The results of the high workload scenario performance test on the maximal deployment are shown in the left part of table 8.3. This table has the same layout as table 8.1, the table denoting the results for the low workload scenario.

Table 8.2: Performance test results for the medium workload variant, before and after optimization with our tool and redeployment of the Microservice Architecture.

Test phase	**Initial MSA**		**Optimized MSA**	
(# requests)	Time (s)	Avg. requests/s	Time (s)	Avg. requests/s
Order (2400)	138	17,3	104	5.74
Article (4800)	272	17,61	212	5.66
Payment (2400)	120	19,89	104	5.74
Delivery (2400)	123	19,41	104	5.73
Review (2000)	102	19,51	88	5.63

Table 8.3: Performance test results for the high traffic variant, before and after optimization with our tool and redeployment of the Microservice Architecture.

Test phase	**Initial MSA**		**Optimized MSA**	
(# requests)	Time (s)	Avg. requests/s	Time (s)	Avg. requests/s
Order (6000)	130	45.83	121	49.30
Article (12000)	237	50.55	230	52.06
Payment (6000)	115	51.88	117	50.96
Delivery (6000)	118	50.60	116	51.43
Review (5000)	66	52.98	69	51.51

Based on the metrics that were emitted during the run, our tool recommends the deployment shown in figure 8.4. This microservice architecture consists of eleven microservices, with a total of 39 duplicated internal feature instances. As can be seen in figure 8.4, several microservices have been merged. According to the simulation performed by our tool, the mean time till consistency increases from 1584 milliseconds to 1612 milliseconds. These numbers indicate that the full parallel processing capacity of the maximal deployment is fully used to handle this workload.

The application was again regenerated and redeployed based on the deployment suggested by our tool. The results of the performance test performed on this architecture are shown in table 8.3. The results of these tests were close to the initial deployment, as shown in table 8.3, however with a lower number of internal feature instances.

8.1.6 Case Study Evaluation

In case of the first workload scenario, the total time of the performance test is reduced with 20% and the throughput of the system increased with 23% on average. Hence, our tool was able to substantially improve the performance of the application. Since this scenario only puts a low workload on the system, a single microservice is able to process all requests without large waiting times. For this workload the overhead of processing every request multiple times by different microservices is larger than the benefits gained by the increased parallel processing.

In the medium workload scenario, the performance was improved substantially as well, although not as much as predicted by the simulation. It is interesting to note that the system has to handle only a quarter of the events compared to the initial microservice architecture. This is a clear result of the fact that now less changes need to be propagated through the system, which also reduces the total amount of events that need to be handled by the microservices.

In the high workload scenario, the performance could not be improved substantially, but the second objective of the fitness function, the duplication of feature instances, could be reduced with 30%, from 55 to 39. As indicated by the simulation, the full parallel processing capacity of the application is required to handle the high workload. Combining several microservices that contain the same internal feature instances, resulted in an substantial decrease of the duplication, without a negative impact on the performance.

The three scenarios show that the different scenarios result in a different deployment, which all were able to substantially improve the overall fitness of the deployment for the defined fitness goals.

9 | Discussion & Opportunities

This chapter first discusses the findings, implications and validity of this research. Secondly the limitations of the current research and opportunities for future research are described.

9.1 Findings and implications

As shown in chapter 8, a redistribution based on the workload resulted in better performance in two cases, and in less duplication in the third case. This shows that it is possible to optimize a microservice architecture by grouping features based on their workload. Since this research has only shown the feasibility of this approach for a single case study, the short-term implications of this research are a call for further research in this area. As discussed in the limitations, it needs to be applied to multiple other cases to verify its generalizability. Secondly as discussed later in this chapter, interesting future work in the area of self-adapting software systems can be derived from this research. In case the success of this approach is confirmed by future work, the long-term implications are that organisations applying this research will benefit from increased performance or decreased hardware costs based on the established goals.

9.2 Validity

This section reflects on the tactics discussed in section 2.6.1 to ensure the validity of this research.

9.2.1 Construct validity

All definitions in this research are based on existing literature, or on common definition of a construct in practice. The performance metrics used in the case study are the result of earlier scientific research of Maddodi et al. (2016) and developers' experience with the system. The constructed fitness objectives are based on existing fields of research and feedback of developers at AFAS, but different organisations might want to use a different weighting of these attributes, or add a custom metric.

9.2.2 Internal validity

The used data collection and analysis methods are described sufficiently to reproduce this research. Secondly all criteria are described in detail. Furthermore all tooling that is not AFAS specific, will be released open-source, so anyone can verify the process. Finally the authors believe they have described every decision in sufficient detail.

9.2.3 External validity

Since this research describes an approach that only requires two context specific components, the objectives of the fitness function and the log input transformer, it should be quite easy to apply the created tooling in another case study company. As mentioned above, the generic part of the tooling will be made open-source, to stimulate other case studies. Secondly Yin (2013) recommends the usage of theory in single-case studies to increase the external validity, therefore the created framework is based on previous research.

9.2.4 Reliability

As mentioned in section 2.6.1, it is relatively easy to reproduce this research since most of the process is automated. The analysis steps are all explained, and thus can be reproduced. However it might take additional effort to reproduce this research in the far future, since several products used in this research are under active development and might change in the future.

9.3 Limitations

As a result of this research being conducted as part of a master thesis, time was a large constraint. As a result of this time constraint, several limitations exist. First of all the workload used in the case study is an artificially created workload, instead of a real workload due to the NEXT still being under development. Secondly due to constraints in the load generator application, the workload had to be altered to become more batch based. As a result of these limitations, it also was not possible to test the queueing theory fitness function, since its assumptions were too heavily violated. The authors however remain confident that this model works, since it has been applied in other researches multiple times with success.

Furthermore due to time constraints, only one base deployment was tested with multiple workloads. Preferably, multiple workloads would have been tested on multiple deployments.

9.4 Opportunities

As a result of the limitations of this research, many interesting opportunities for further research exist. Further research into the robustness of the genetic algorithm is required, such as robustness against small variations in workload, and reducing the effect of non-determinism in the algorithm. More case studies are essential to further optimize our approach, preferably with a real production workload, to confirm the outcome of this research. Furthermore we see opportunities for a (quantitative) study to determine whether the proposed optimizations result in an actual reduction of operational cost in practice.

9.4.1 Fitness objectives

While several objectives have been defined and implemented in the case study, more candidate objectives exists. Several options for fitness objectives, that are worth analysing according to the authors, are given below:

Standard deviation of requests between features in a microservice The idea behind this objective is that microservices with features that are called equally, result in less unnecessarily scaled features. For example assume feature A and B are both located in the same microservice and feature A processes 1 million requests in an hour, while feature B only processes 10 requests in an hour. In case this microservice is scaled, because of the many requests to A, B is also scaled, which is absolutely unnecessary. By creating microservices of features that process around the same amount of requests, the amount of unnecessary scaling should be reduced.

Pattern based objectives In a real workload, users typically follow patterns. For example they first create an order, followed by an invoice. This also results in a pattern in the requests, which can be used to optimize the deployment. In case a user typically performs two, for the system unrelated, actions after each other, these features should be located in different microservices.

9.4.2 Different perspectives

This research focussed on optimizing microservices with regards to performance and data-duplication. The effect of these optimization on other quality attributes can be examined. Secondly the feature grouping of microservice architectures can also be optimized for other quality attributes, such as security and maintainability, instead of performance.

9.4.3 Self adapting software

A self-adaptive system observes its own behavior and analyzes these observations to determine appropriate adaptation (Oreizy et al., 1999). Salehie and Tahvildari (2009) state that such a system must monitor itself and its context, detect significant

changes, decide how to react and act to execute such decisions. A major property of self adapting systems is the ability to be self-optimizing according to the hierarchy created by Salehie and Tahvildari (2009).

In case a generated software application is used, such as Profit Next, the created tooling can be used to create a continuous feedback loop that alters the application based on the workload at hand. In case the workload changes significantly, the created microservice deploymenet can be automatically altered to optimize the application based on the defined goals. As a result, the application becomes self-optimizing based on the workload at hand.

10 | Conclusions

This chapter will answer the research questions of this master thesis.

SRQ 1 | *Which metrics are relevant for grouping features of microservice architectures?*

At the lowest level, performance metrics such as the service time and waiting times of microservices and individual features are required. As discussed in chapter 6, several higher level metrics are relevant for grouping features of microservice architectures. The average sojourn time, the average waiting time and service time combined, is an important metric in case of event-driven microservices architectures. Since they use asynchronous messaging between microservices to propagate changes, an increased average sojourn time results in slower propagation of these changes. In such a case it is easy to see that a user is more likely to see an inconsistent state of the system, by viewing data from an internal feature instance that has not processed the latest change yet. Hence it is also desired to keep the sojourn time as low as possible from a usability perspective. Furthermore the utilization of a microservice is an interesting metric from a cost perspective, since a more efficient usage of the available capacity results in less unused capacity, resulting in less hardware. The duplication of features and data is a relevant metric from a cost and maintenance perspective. Less duplication of data results in less hardware requirements, since less data needs to be stored and processed. Furthermore less duplication of code improves the maintainability of the application.

SRQ 2 | *How can metrics be linked to features?*

A common requirement to link metrics to features is the option to add meta-data to a metric. Hence it is recommended to use a logging and monitoring platform that supports this. Depending on the type of metric, another strategy has to be used. Three categories of metrics can be distinguished: application metrics, platform metrics and system metrics. Application metrics are metrics reported by the application itself. Since these metrics are created by the developers of the application, it is easy to add tracing meta-data to link the metric to an individual feature. In case of platform metric, metrics reported by the underlying framework, it becomes a bit harder. Depending on the metric, feature location techniques can be used to determine the related feature. In most cases it is not possible to relate system metrics directly to features, since most system metrics are typically available on process level. In many

implementations of a microservice architecture, a process contains many features, making it impossible to identify which feature is responsible for the metric. In some cases it might be deducible by comparing the system metric with application metrics. Based on the microservice deployment and the performance metrics, SRQ 4 can be answered:

SRQ 3 | *What is an effective way to find improvements in the grouping of features in a microservice architecture?*

The fitness function described in chapter 6 can be used to evaluate the performance of a particular microservice deployment. This fitness function is combined with a genetic algorithm, as described in chapter 7, to search for efficient deployments of a set of features based on the objectives of the fitness function. Based on the fitness function, the algorithm quickly stops searching in areas of the search space that do not contain more efficient deployments.

SRQ 4 | *What is the effect of the architecture modifications on scalability and performance?*

The suggested applications result in an architecture that is more tailored towards the workload at hand. This is clearly shown in the case study described in chapter 8, in which a single application is adapted differently for three different workloads. It is thus important to have a representative workload. In general a merge of a microservice results in lower throughput of that microservice, but in cases where the reduced throughput is sufficient to handle the workload, it becomes faster since of the reduction in propagation messages, as shown in the medium workload scenario of the case study. However, the total throughput capacity of the application should be sufficient for the workload at hand. In general having the microservices as small as possible results in more parallel capacity, but also puts more load on the system, since all changes need to be propagated to multiple other microservices.

RQ | *How to group features based on application workload to improve performance of a microservice architecture?*

This question is answered by the created tooling as part of this research. It only needs a microservice deployment model as described in chapter 4 and a workload definition with the corresponding performance metrics. The tooling only requires a case specific workload adapter to be able to analyse the current deployment and suggest improvements based on the fitness function. In case a company has specific optimization objectives, it is easy to alter the fitness function to take this objective into account.

Bibliography

Aalst, W. V. D. (2012). Process mining: Overview and opportunities. *ACM Transactions on Management Information*.

Abbott, M. L. and Fisher, M. T. (2009). *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. Addison-Wesley Professional.

Allen, A. (2014). Probability, statistics, and queueing theory.

Bass, L., Clements, P., and Kazman, R. (2007). Software architecture in practice.

Braddock, R. L., Claunch, M. R., Rainbolt, J. W., and Corwin, B. N. (1992). Operational Performance Metrics in a Distributed System: Part II -Metrics and Interpretation. *Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing: technological challenges of the 1990's*, pages 873–882.

Brewer, E. (2012). CAP twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29.

Brewer, E. A. (2000). Towards robust distributed systems. In *PODC*.

Cecchet, E., Chanda, A., and Elnikety, S. (2003). Performance comparison of middleware architectures for generating dynamic web content. *Proceedings of the ....*

Crockford, D. (2006). The application/json media type for javascript object notation (json).

Dit, B., Revelle, M., and Gethers, M. (2013). Feature location in source code: a taxonomy and survey. *Journal of Software:*.

Fowler, M. (2014). Microservices. http://martinfowler.com/articles/microservices.html.

Fowler, M. (2015). Microservice Trade-Offs. http://martinfowler.com/articles/microservice-trade-offs.html.

Fulton, S. (2015). What Led Amazon to its Own Microservices Architecture - The New Stack. http://thenewstack.io/led-amazon-microservices-architecture/.

Geitgey, A. (2013). I-Tier: Dismantling the Monolith | - Groupon Engineering Blog. https://engineering.groupon.com/2013/misc/i-tier-dismantling-the-monoliths/.

Google Trends (2015). Google Trends - Web Search interest: microservices - Worldwide, 2004 - present. https://www.google.com/trends/explore?hl=en#q=microservices.

Guelen, J. (2015). Informed CQRS design with continuous performance testing.

Harman, M., Hierons, R. M., and Proctor, M. (2002). A New Representation And Crossover Operator For Search-based Optimization Of Software Modularization. In *GECCO*, pages 1351–1358.

Hevner, A. R., March, S. T., Park, J., and Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, 28(1):75–105.

Holland, J. (1975). Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.

Kabbedijk, J., Jansen, S., and Brinkkemper, S. (2012). A case study of the variability consequences of the CQRS pattern in online business software. In *Proceedings of the 17th European Conference on Pattern Languages of Programs - EuroPLoP '12*, pages 1–10, New York, New York, USA. ACM Press.

Lavenberg, S. (1983). Computer performance modeling handbook.

Leppanen, M., Makinen, S., Pagels, M., Eloranta, V.-P., Itkonen, J., Mantyla, M. V., and Mannisto, T. (2015). The Highways and Country Roads to Continuous Deployment. *IEEE Software*, 32(2):64–72.

Levy, R., Nagarajarao, J., Pacifici, G., Spreitzer, A., Tantawi, A., and Youssef, A. (2003). Performance management for cluster based Web services. In *IFIP/IEEE Eighth International Symposium on Integrated Network Management, 2003.*, pages 247–261. Kluwer Academic Publishers.

Maddodi, G., Jansen, S., Guelen, J., and de Jong, R. (2016). The Daily Crash: A Reflection on Continuous Performance Testing. *ICSEA 2016*.

Mahdavi, K., Harman, M., and Hierons, R. (2003). A multiple hill climbing approach to software module clustering. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 315–324. IEEE Comput. Soc.

Mancoridis, S., Mitchell, B., Chen, Y., and Gansner, E. (1999). Bunch: a clustering tool for the recovery and maintenance of software system structures. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 50–59. IEEE.

Mancoridis, S., Mitchell, B., Rorres, C., Chen, Y., and Gansner, E. (1998). Using automatic clustering to produce high-level system organizations of source code. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*, pages 45–52. IEEE Comput. Soc.

Marchini, J. (2008). Lecture 5 : The Poisson Distribution.

Mauro, T. (2015). Microservices at Netflix: Lessons for Architectural Design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

Meier, J., Vasireddy, S., Babbar, A., Mariani, R., and Mackman, A. (2004). Chapter 15 - Measuring .NET Application Performance. https://msdn.microsoft.com/en-us/library/ff647791.aspx.

Menouar, B. (2010). Genetic algorithm encoding representations for graph partitioning problems. In *2010 International Conference on Machine and Web Intelligence*, pages 288–291. IEEE.

Oreizy, P., Gorlick, M., and Taylor, R. (1999). An architecture-based approach to self-adaptive software. *... Intelligent systems*.

Overeem, M., Spoor, M., and Jansen, S. (2017). The Dark Side of Event Sourcing: Managing Data Conversion. In *SANER 2017*. IEEE Comput. Soc.

Polya, G. (2014). How to solve it: A new aspect of mathematical method.

Praditwong, K., Harman, M., and Yao, X. (2011). Software Module Clustering as a Multi-Objective Search Problem. *IEEE Transactions on Software Engineering*, 37(2):264–282.

Richardson, C. (2014). Event-driven architecture.

Rob Ewaschuk, B. B. (2016). Monitoring Distributed Systems. 1.

Rubin, J. and Chechik, M. (2013). A survey of feature location techniques. *Domain Engineering*.

Salehie, M. and Tahvildari, L. (2009). Self-adaptive software: Landscape and research challenges. *... on Autonomous and Adaptive Systems (TAAS)*.

Stenberg, J. (2014). Martin Fowler on Characteristics of Microservices. http://www.infoq.com/news/2014/11/gotober-fowler-microservices.

Swartout, P. (2014). *Continuous Delivery and DevOps âĂŞ A Quickstart Guide - Second Edition*.

Thoughtworks (2014). Microservices | Technology Radar | ThoughtWorks. https://www.thoughtworks.com/radar/techniques/microservices.

Urgaonkar, B., Pacifici, G., Shenoy, P., and Spreitzer, M. (2005). An analytical model for multi-tier internet services and its applications. *ACM SIGMETRICS*.

van der Schuur, H., Jansen, S., and Brinkkemper, S. (2011). Reducing Maintenance Effort through Software Operation Knowledge: An Eclectic Empirical Evaluation. In *CSMR 2011*, pages 201–210.

van der Werf, J. M. E. M. and Verbeek, H. M. W. (2015). Online Compliance Monitoring of Service Landscapes. In *BPM Workshops, Revised Papers*, volume 202, pages 89–95.

Vogels, W. (2009). Eventually consistent. *Communications of the ACM*, 52(1):40.

Whitley, D. (1994). A genetic algorithm tutorial. *Statistics and computing*.

Woodside, M., Franks, G., and Petriu, D. C. (2007). The Future of Software Performance Engineering. In *Future of Software Engineering (FOSE '07)*, pages 171–187. IEEE.

Wootton, B. (2014). Microservices - Not a free lunch! - High Scalability -. http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html.

Yin, R. K. (2013). *Case Study Research: Design and Methods*. Sage publications.

Appendices

A | Case study results

This appendix shows the individual test results for the three different workload scenarios.

A.1 Low workload scenario

Table A.1 shows the aggregated performance metrics for the individual tests per phase of the test. In the order phase, all orders are inserted in the system. Afterwards the products are insert. In the payment phase, 600 payment requests are performed. In the delivery phase, 600 deliveries are created linked to the orders. Finally in the review phase, 500 reviews of products are created. The setup phases of users and products are not listed in this table. The failed requests column represents the number requests that returned an error code. The total time represents the total time in seconds it took for each phase to complete. The requests per second columns represent the average, minimum and maximum throughput respectively. The handletime columns represents the average, minimum and maximum service time of the microservices. The Eventbus waiting time represents the time the systeem required to catch up after all requests were completed. Finally the 'Max items in eventbus' column describes how many events there were maximally during the catch-up.

Test		Failed	Total time (s)	Requests/second			Handletime (ms)			Eventbus	Max. items
Subtest	run #	requests		avg	min	max	avg	min	max	waitingtime (s)	in eventbus
Order (600 request)	1	0	117	5,11	3	8	66	28	393	0	0
	2	0	141	4,24	0	7	106	30	3269	0	0
	3	0	124	4,81	2	7	75	29	472	0	0
	4	0	131	4,56	0	7	83	33	1677	0	0
	5	0	142	4,24	0	7	102	29	22406	0	0
Products (1200 requests)	1	0	215	5,56	3	7	48	29	392	0	0
	2	0	254	4,71	0	8	81	29	3449	0	0
	3	0	253	4.74	0	8	78	30	1465	1	5
	4	0	270	4,44	0	7	94	29	2987	0	0
	5	0	293	4,09	0	7	110	28	21850	0	0
Payment (600 requests)	1	0	104	5,74	3	8	40	28	1035	0	0
	2	0	127	4,71	0	7	83	27	3358	2	1
	3	0	129	4,63	0	8	80	29	2092	0	0
	4	0	132	4,53	0	7	88	27	2766	0	0
	5	0	124	4,82	0	7	75	29	2634	0	0
Delivery (600 requests)	1	0	107	5,58	3	8	44	30	331	0	0
	2	0	127	4,7	0	7	78	30	2067	0	0
	3	0	125	4,78	0	7	73	32	1891	0	0
	4	0	126	4,74	0	8	77	31	3061	0	0
	5	0	160	3,74	0	8	135	31	22849	0	0
Review (500 requests)	1	0	89	5,59	3	8	43	28	251	0	0
	2	0	101	4,93	0	7	69	29	2019	0	0
	3	0	101	4,93	0	8	68	29	2402	2	0
	4	0	108	4,61	0	7	85	29	4159	0	0
	5	0	106	4.7	0	7	80	29	1700	3	39

Table A.1: Aggregated results of the low workload tests

A.2 Medium workload scenario

Table A.2 shows the aggregated performance metrics for the individual tests per phase of the test under a medium traffic workload. A description of the individual columns can be found in section A.1.

A.3 High workload scenario

Table A.3 shows the aggregated performance metrics for the individual tests per phase of the test under a high traffic workload. A description of the individual columns can be found in section A.1.

Test		Failed	Total time (s)	Requests/second			Handletime (ms)			Eventbus	Max. items
Subtest	run #	requests		avg	min	max	avg	min	max	waitingtime (s)	in eventbus
Order (2400 request)	1	0	138	17,3	2	26	90	36	3560	38	225
	2	0	144	16,58	1	25	101	37	3599	38	185
	3	0	135	17,68	2	24	92	34	3031	14	300
	4	0	125	19,08	3	23	76	35	513	0	0
	5	0	143	16,71	0	24	99	35	3782	0	0
Article (4800 requests)	1	0	272	17,61	1	25	87	32	1369	41	380
	2	0	253	18,95	0	24	76	31	3467	110	1041
	3	0	262	18,29	2	24	83	33	5071	19	89
	4	0	257	18,65	2	26	81	34	1055	82	643
	5	0	300	15,97	1	25	113	32	3269	11	130
Payment (2400 requests)	1	0	120	19,89	1	25	63	31	574	49	326
	2	0	127	18,8	3	25	76	30	2687	5	10
	3	0	147	16,22	5	25	112	35	3030	24	288
	4	0	139	17,18	3	25	93	31	3941	117	832
	5	0	135	17,67	3	27	86	31	2288	46	564
Delivery (2400 requests)	1	0	123	19,41	4	24	70	35	389	74	613
	2	0	136	17,56	2	23	90	37	1454	59	480
	3	0	126	18,95	6	27	78	36	527	83	637
	4	0	129	18,51	6	24	84	34	2013	84	748
	5	0	125	19,11	1	26	72	37	457	21	197
Review (500 requests)	1	0	102	19,51	2	25	68	35	629	164	989
	2	0	114	17,47	1	25	89	33	1139	125	1014
	3	0	137	14,53	1	24	132	35	1999	161	1125
	4	0	127	15,65	3	24	113	31	2276	165	1062
	5	0	102	19,49	4	25	70	33	478	106	873

Table A.2: Aggregated results of the medium workload tests

Test		Failed	Total time (s)	Requests/second			Handletime (ms)			Eventbus	Max. items
Subtest	run #	requests		avg	min	max	avg	min	max	waitingtime (s)	in eventbus
Order (6000 request)	1	0	130	45,83	1	56	81	37	1077	334	5037
	2	0	128	46,59	7	55	77	38	426	241	3828
	3	0	124	48,06	3	61	75	37	332	327	5042
	4	0	127	46,94	4	58	76	36	1090	280	3631
	5	0	133	44,82	2	53	86	38	1112	327	3796
Article (12000 requests)	1	0	237	50,55	1	65	63	33	1072	348	10750
	2	0	240	49,92	11	60	66	32	1055	431	9323
	3	0	242	49,51	5	61	66	31	1058	352	9659
	4	0	240	49,93	1	61	64	34	3121	4115	9118
	5	0	239	50,11	10	60	66,0	32	386	387	9459
Payment (6000 requests)	1	0	115	51,88	3	64	56	29	3059	156	3782
	2	0	117	50,99	3	62	62	30	3189	153	4043
	3	0	114	52,31	23	66	57	29	283	155	3733
	4	0	117	50,97	1	66	58	30	3055	154	3578
	5	0	111	53,73	15	66	55	31	398	132	3909
Delivery (6000 requests)	1	0	118	50,6	12	61	64	33	1074	123	3811
	2	0	117	50,96	1	70	61	34	365	154	4006
	3	0	119	50,12	38	62	63	35	295	154	4248
	4	0	118	50,56	12	60	62	33	358	128	3921
	5	0	120	49,7	2	60	63	33	373	153	4149
Review (5000 requests)	1	0	66	52,98	35	65	59	32	376	182	2854
	2	0	67	52,24	9	62	58	31	359	181	2749
	3	0	70	50,04	35	62	63	35	365	202	2767
	4	0	68	51,44	10	63	61	32	311	179	2801
	5	0	67	52,54	40	63	58	30	355	183	2753

Table A.3: Aggregated results of the high workload tests

B | Paper

Workload-based Clustering of Coherent Feature Sets in Microservice Architectures

Sander Klock*†, Jan Martijn E. M. van der Werf*
, Jan Pieter Guelen† and Slinger Jansen*
* Utrecht University, Princetonplein 5, 3584 CC Utrecht, Netherlands
Email: {j.m.e.m.vanderwerf, slinger.jansen}@uu.nl
†AFAS Software, Philipsstraat 9, 3833 LC Leusden, Netherlands
Email: {s.klock, j.guelen}@afas.nl

***Abstract*—In a microservice architecture, each service is designed to be independent of other microservices. The size of a microservice, defined by the features it provides, directly impacts performance and availability of the microservice. However, none of the currently available approaches take this into account. This paper proposes an approach to improve the performance of a microservice architecture by workload-based feature clustering. Given a feature model, the current microservice architecture, and the workload, this approach recommends a deployment that improves the performance for the given workload using a genetic algorithm. We created MicADO, an open-source tool, in which we implemented this approach, and applied it in a case study on an ERP system. For different workloads, the resulting generated microservice architectures show substantial improvements, which sets the potential of the approach.**

I. Introduction

Interest in microservice architectures has increased over the last few years, with a significant increase since 2014 [24]. A microservice architecture is an architecture in which a single application is designed as a set of independent small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP API [16]. As a result of this, every module is an independently deployable service. Combined with the lightweight communication protocols used, every service can use its own programming language and can be easily modified and scaled.

The size of a microservice is directly defined by its features, i.e., chunks of functionality that deliver business value [4]. A microservice that offers more features will be larger than a service with only a few features. The term microservice indicates that services should be small. However, people are reluctant to define how small they should be [23]. There are several metrics for the size of microservices, such as lines of code of a microservice, being able to rewrite a microservice in 6 weeks or having a 2-pizza team (two pizzas are enough to feed the entire team) per service [5]. Another typical answer is that a microservice should do one thing, which leaves room for interpretation.

None of the existing metrics are related to quality attributes [2]. However, the size of a microservices has a direct impact on the performance and scalability of the application. As a result of this observation, metrics related to performance and scalability seem more appropriate than the existing metrics.

Moving features to other or new microservices directly impacts the performance and scalability of the system. The size of the smallest scalable unit becomes smaller, resulting in an increase of scalability. The effect on the performance of the system however depends on the relationship between its features. If, for example, two features are heavily dependent on each other, splitting them over different microservices might result in significant communication overhead, and thus performance decreases. Merging two microservices results in a loss of scalability, but performance might increase due to decreased communication overhead. Additionally, the actual usage of features by users determines the impact of moving features. If a seldom-used feature is moved, the impact is much smaller than moving a feature that is frequently being used.

Based on these observations, this paper proposes an automated approach for optimizing the performance and scalability of a microservice architecture by modifying the placement of features in microservices based on the workload of a microservice system. The approach is depicted in Figure 1. Based on a feature model that describes the properties and dependencies of the features the architecture should implement, the software operation data collected as result of a workload on the current system, our approach suggests a clustering of these features in microservices, optimized for the given workload.

The remainder of this paper is structured as follows. Section II introduces the feature model and its mapping to microservice architectures, and Section III describes how we

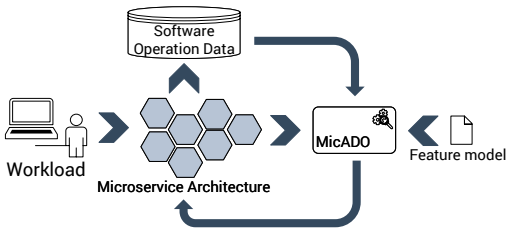


Fig. 1: Overall overview of the proposed approach.

This is an AMUSE paper. See amuse-project.org for more information.

measure the workload of a running system. The feature model and workload are input of our genetic optimization algorithm, which is discussed in Section IV. Our approach is validated in a case study, of which the results are presented in Section V. Finally Section VI provides a discussion of this research and concludes the paper.

II. Modeling Features in Microservices

In this section, we introduce the Feature Model and a model for Microservice Architectures.

A. Feature Model

The heart of the microservice model is formed by the set of features that the system should implement. In a microservice architecture, these features are distributed over different microservices. This distribution is influenced by the dependencies between the different features. Ideally, all depending features will be placed together in a single microservice. A feature is represented by a set of unique properties together with a set of features it depends upon. Since many formalisms exist to express features and their dependencies, such as Feature Diagrams [20], we only formalize the elements that are required, so that architects can freely choose their favorite notation. In our formalization, the set of features F is a partitioning of the set of properties P, i.e., each property belongs to exactly one feature. Similarly, the properties of the feature determine the feature dependencies. We therefore model the feature dependencies as a directed graph on the properties. This results in the following definition of a Feature Model:

Definition II.1 (Feature Model)
A feature model is a 3-tuple (P, F, R) with

- a set of *properties* P;
- a set of *features* F, being a partitioning of P;
- and the *dependency graph* (P, R), a directed graph.

Note that we allow properties to depend on properties within the same feature. An example feature model is depicted in Figure 2. This example has three features, A, B, and C, with three properties each. Property P_3 of feature A depends on both property P_4 of B and property P_7 of C. Similarly, property P_4 and P_5 of feature B depend on properties P_8 and P_9 of C, respectively. In our model, we thus represent feature A by $\{P_1, P_2, P_3\}$, and the set of all features F by $\{\{P_1, P_2, P_3\}, \{P_4, P_5, P_6\}, \{P_7, P_8, P_9\}\}$.

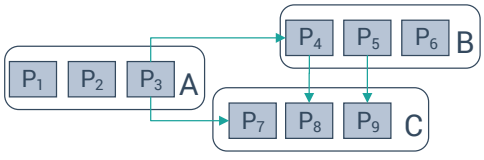


Fig. 2: Example feature model with three features, A, B and C, each consisting of three properties.

B. Microservice Architectures

A microservice architecture implements a feature model, by instantiating features in microservices. A first approach would be to partition the features over the different microservices. Although this would correctly model microservice architectures, it is not sufficient to model event-driven microservices [19]. In this model, features publish updates to which other features can subscribe, which results in a cascade of feature updates, and thus in more communication between the microservices. A common practice to avoid this cascading effect, is to duplicate dependent features within a single microservice. These duplicated features are internal, i.e., only available within that microservice. The feature emitting the events is the only feature that exposes that functionally publicly, and contains all properties. This feature instance is the *public feature* instance of a feature. Every feature thus has at least one public feature instance and zero or more internal feature instances. An internal feature contains a non-empty subset of the properties of the feature, since it might only require a part of the data from an event. This results in the following definition of a Microservice Architecture Model:

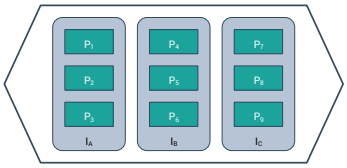
Definition II.2 (Microservice Architecture)
Given a feature model (P, F, R), a *Microservice Architecture* is a 4-tuple (I, M, λ, h) with:

- a set of *feature instances* I;
- a set of *microservices* M, being a partitioning of I;
- the *property instantiation function* $\lambda : I \to \mathbb{P}(P)$, a total function that maps each feature instance to a set of properties;
- the *public instance function* $h : F \to I$, a total function that defines for each feature its public instance;

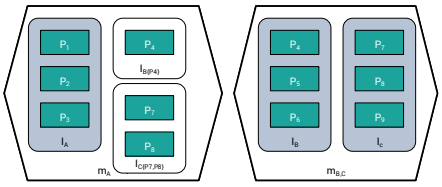
such that

- Each microservice contains all instances necessary to fulfil the dependency requirements, i.e.
  $\forall m \in M\colon \forall i \in m, p \in \lambda(i), q \in P :$
  $(p, q) \in R \implies \exists j \in m : q \in \lambda(j)$
- Every microservice contains each feature at most once, i.e.
  $\forall m \in M\colon \forall i, j \in m : \exists f \in F :$
  $(\ \lambda(i) \subseteq f \wedge \lambda(j) \subseteq f\ ) \implies i = j$
- Each feature instance is a subset of its feature, i.e.
  $\forall i \in I : \exists f \in F : \lambda(i) \subseteq f$
- Each feature has a public instance that is equal to itself, i.e.,
  $\forall f \in F, i \in I : h(f) = i \implies f = \lambda(i)$
- Each microservice contains at least one public feature instance, i.e.
  $\forall m \in M : \exists i \in m, f \in F : \lambda(i) = h(f)$

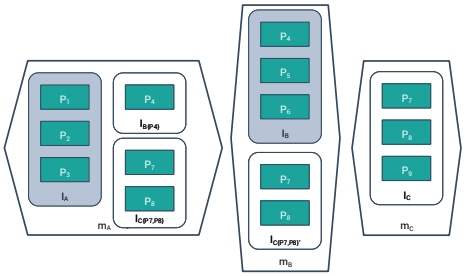
Consider again the example feature model of Figure 2. The simplest deployment for this feature model would be to create a microservice architecture in which all features are instantiated in a single microservice, as depicted in Figure 3(a). We call this architecture the minimal microservice architecture. The gray elements indicate public feature instances, while the white elements indicate internal feature instances. Another possibility would be to deploy a microservice architecture where each microservice has exactly one public feature, called the maximal microservice architecture. In this case, there will be three microservices, m_A, m_B and m_C. As a result of the dependencies defined in the feature model, this introduces internal feature instances in the microservices. We denote a feature instance by $i_{X\{P_1,\ldots,P_n\}}$, where $P_1,\ldots,P_n$ are properties of feature X. We omit the subset of properties if it is the complete set of properties of that feature. For microservice m_A, this results in $m_A = \{i_A, i_{B\{P_4\}}, i_{C\{P_7,P_8\}}\}$. The other microservices can be represented as $m_B = \{i_B, i_{C\{P_8,P_9\}}\}$ and $m_C = \{i_C\}$.



(a) Minimal deployment



(b) Mixed deployment



(c) Maximal deployment

Fig. 3: Possible Microservice Architectures for the feature model depicted in Figure 2.

III. Measuring workload through Software Operation Data

The second component of our approach is the workload of a deployed architecture. We define the workload in terms of concurrent users and used features as a function of time. Time is an important dimension in the usage of an application.

One way to obtain the workload of a deployed microservice architecture is by monitoring its operation. Monitoring the operation of a system is not new and the use of System Operation Data [21, 26] is widely used in software engineering practices [3], such as maintainability [22], problem diagnosis [27] and compliance [26]. In the remainder of this section, we apply software operation data to obtain both the usage and the performance of a deployed microservice architecture.

A. Feature Usage

Feature usage over time provides valuable insight in frequent usage patterns, and therefore are worth optimizing for. Usage over time provides valuable insight in peak usage, while this is lost if aggregations, such as the mean, would be used. In the case of a microservice architecture, most communication is performed via lightweight mechanisms, such as the HTTP protocol, and support logging out of the box. In essence, a microservice architecture follows the client-server paradigm, where clients interact by requesting services of servers, which provide a set of services [2]. From the access log, which contains the information which feature has been called, by whom and when, it is possible to derive the usage of features at the server. Process Mining [1], a set of tools and techniques to discover, monitor and improve real processes by extracting knowledge from event logs, allows us to analyze these access logs to derive relevant feature usage metrics.

B. Performance Metrics

Feature usage is only one aspect of workload of a system. Although access logs provide useful insights in feature usage, they do not contain any information about the actual system performance and scalability. Metrics are able to provide this insight. We define a metric as a more abstract representation, such as the mean or sum, of a time series of individual measurements. A measurement is a quantitative attribute of a running software system that can be measured automatically. Again these metrics can be derived from system operation data, e.g. by monitoring when a microservice executes a feature, and its duration. Based on these measures, performance metrics can be derived. An important requirement for all performance metrics is that all metrics should allow to be traced back to the indivual features a microservice implements. Different levels of performance metrics can be identified [14]:

Application metrics Application metrics are metrics reported by the application itself. These metrics are specific for the application and are typically related to significant events in the domain of the application. An example of such a metric might be the number of products ordered per minute for a webshop.

Platform metrics Platform metrics are metrics reported by the framework or runtime of the application. These metrics are related to significant events occurring in the runtime of the application. For example the Microsoft .Net platform reports the number of exceptions thrown per second [14].

System metrics System metrics are reported by the operating system and/or hardware of the server. These metrics provide information about significant events on the hardware level. The number of CPU interrupts per second is an example of a system metric.

Platform and system metrics typically have process level as smallest granularity level. This means that common metrics such as memory usage are available for individual processes. A finer level of detail can be achieved using profilers, however they have a significant impact on the performance of an application, making them not suitable for production environments. While process level granularity typically provides sufficient details for monitoring a running application, it does not provide sufficient detail to link them to individual features, since every processes contains one or multiple features.

Typically these metrics are linked to features by adding metadata to the metric. It is recommended to use a logging and monitoring system that support structured metrics, to support metadata, and thus the ability to link them to individual features. If the metadata contains both a unique request identifier and feature identifying data, the performance impact of the system's usage per feature can be determined using performance or process mining tools.

IV. Optimization Algorithm

Now that both the deployment and the workload have been described, the deployment can be improved based on the workload.

The problem at hand, the distribution of features over microservices, is closely related to the problem of software module clustering. Software Module Clustering is defined as automatically finding a good clustering of software modules based on the relationships among the modules [11]. In this field, several optimization approaches have been proposed, such as hill climbing [12, 13] and genetic algorithms [7, 18]. Both methods use a fitness function to express the quality of the clustering. Since the approach using an genetic algorithm combined with a multi-objective approach by [18] resulted in better results than hill climbing, we decided to use a genetic algorithm to solve this problem.

A. Genetic Algorithm

In order to apply a genetic algorithm to solve a problem, the problem should be genetically encodable, such that the genetic operators mutation and crossover are able to transform a chromosome in a meaningful way. The genetic encoding is a representation of the problem that resembles the way DNA is represented. Typically this is depicted as an array of bits or characters.

A single chromosome in the population should represent a single microservice architecture. As described in Section II, a microservice architecture can be described as a 4-tuple (I, M, λ, h) given a feature model. Chromosomes should be encoded in such a way that it is possible to compare them.

Different microservice architectures of the same feature model, contain a different number of internal feature instances. For example, the minimal microservice architecture contains no internal feature instances, whereas the maximal microservice architecture contains the most feature instances. Feature instances are thus sub-optimal to represent a microservice architecture. A possible solution would be to include all feature instances in the encoding of a deployment. This would result in overhead in the representation, as many feature instances are not present in a deployment.

However, the feature instances required for a deployment can be derived from the placement of the features over the microservices and the dependency graph of the feature model. This dependency graph encodes which features should be created internally to obtain an independent microservice where all dependent features are included. As the set of features of a feature model is stable across the deployments, this representation is an efficient encoding of the problem.

A simple representation of the placement of a feature in a microservice is to assign an integer to every feature that represents in which cluster it is located. As an example, for the microservice architecture depicted in Figure 3(b), mapping feature A to 1, B to 2 and C to 3, and microservice m_A to 1 and $m_{B,C}$ to 2, results in the encoding shown in Table I.

TABLE I: Genetic encoding of a deployment

Feature	1	2	3
Microservice	1	2	2

However this simple representation does not uniquely identify a deployment, as shown by Table II, which gives another encoding of the same deployment. A solution having more than one representative chain in the encoding scheme results in the encoding having redundancy [15]. The redundancy of this simple encoding is large, since a deployment with m microservices, can be represented by $m!$ different chains. Since the redundancy grows exponentially for the number of microservices, a large part of the domain of the genetic encoding consists of duplicate chains.

TABLE II: Different genetic encoding of a deployment shown in table I

Feature	1	2	3
Microservice	2	1	1

To solve this problem, the microservice identifier should be deterministically derived in such a way that the same features in a microservice result in the same microservice identifier. Hence the identifier should be based on the features contained in the microservice. Instead of assigning an incremental integer as identifier of the cluster, the numeric identifier of the feature

with the highest feature number is chosen as microservice identifier. An example of this encoding applied to the same deployment is shown in Table III.

TABLE III: Non redundant genetic encoding of a deployment shown in Table II

Feature	1	2	3
Microservice	1	3	3

B. Fitness Function

To compare several deployments, a function that expresses the quality of a deployment is required. The fitness of a deployment can be calculated by actually implementing the deployment and executing the workload on the deployment. Based on the performance metrics, the fitness can be determined. However, evaluation of a deployment becomes cumbersome and time consuming. Hence approximations of the performance of a deployment are required.

Queueing networks is a well-established method for performance modelling [9]. Since computer systems can be represented as (networks of) queues and servers, this is a popular performance modelling technique.

The simplest model of queueing theory, the M/M/1 model, has proven useful in several real life scenarios. For example, it has been used in the performance analysis of cluster-based web services [10] and multi-tier internet services [25].

Extensions to this model are required to model a microservice architecture. The standard M/M/1 model assumes that all requests are of the same type, and have an exponential distribution. A microservice containing several features, will process requests of different types. Based on the type of requests, it is likely that different types of requests have a different exponential distribution. As in a M/M/1 model, the arrival rate is a Poisson process which can be merged and split [9] based on the chance that a request has a certain class. Thus, each class has its own service rate, denoted by μ_i, arrival rate, denoted by λ_i, and a chance of occurring, denoted by p_i. In this model, the total arrival rate of the server is:

$$\lambda = \sum_{i=1}^{n} (\ p_i \cdot \lambda_i \ ) \tag{1}$$

i.e. the total arrival rate of the service equals the weighted sum of the arrival rates of the different customer classes. Similarly, the service time is calculated as the weighted sum of the service times of the different classes. Hence, the mean service time of the server can be calculated using the following formula:

$$\mu = \sum_{i=1}^{n} (\ p_i \cdot \lambda_i \cdot \mu_i \ ) \tag{2}$$

The formalization of the utilization and waiting times remain the same as for the M/M/1 case, i.e., the utilization is defined by $\rho = \frac{\lambda}{\mu}$.

As an approximation for the chance that a request is of a certain class, we assume the classes to be uniformly distributed, i.e.,

$$p_i = \frac{\lambda_i}{\sum_{j=1}^{n} \lambda_i} \tag{3}$$

Similarly, the arrival rate, service rate and waiting time need to be approximated for the deployments under evaluation by the genetic algorithm.

For brevity the following notation is introduced:

Definition IV.1 (Microservice Performance Model)
A microservice m is a 2-tuple (λ, μ), where λ denoted the mean arrival rate, and μ denotes the mean service time.

Now, a merge of two microservices $m_A = (\lambda_A, \mu_A)$ and $m_B = (\lambda_B, \mu_B)$ can be represented by

$$m_i \oplus m_j = (\lambda_{merge}, \mu_{merge}) \tag{4}$$

It is trivial to see that λ_{merge} is the sum of the arrival rates of the individual microservices, i.e., $\lambda_{merge} = \lambda_A + \lambda_B$. Unfortunately, calculating the mean service rate is more complicated. Summing the individual service times coincides running them in parallel, which is clearly not the case in microservices. Hence the mean of both μ_A and μ_B seems more appropriate, i.e. the mean service rate of the merged microservice is the weighted mean service rate of both individual microservices. It is easy to think of a scenario in which λ_A is much larger than λ_B. μ_{merge} would be skewed towards μ_A with a simple mean. Hence a weighted mean based on the arrival rate of the different customer types seems more appropriate. This results in the following formula:

$$\mu_{merge} = \frac{\lambda_A}{\lambda_A + \lambda_B} \cdot \mu_A + \frac{\lambda_B}{\lambda_A + \lambda_B} \cdot \mu_B \tag{5}$$

Based on the calculated mean arrival rate and the mean service rate, the waiting time can be calculated using the same formula for simple M/M/1 queues.

Definition IV.2 (Merging two microservices)
Given two microservices $m_A = (\lambda_A, \mu_A)$ and $m_B = (\lambda_B, \mu_B)$, their merge, denoted by $m_A \oplus m_B$ is again a microservice, defined by

$$m_A \oplus m_B = (\lambda_A + \lambda_B, \frac{\lambda_A}{\lambda_A + \lambda_B} \cdot \mu_A + \frac{\lambda_B}{\lambda_A + \lambda_B} \cdot \mu_B)$$

C. Assumptions and Approximations

Approximating a microservice by a queueing server is only possible under certain assumptions. First, we assume requests to arrive memoryless, i.e., the inter-arrival rate between two requests is independent. Additionally, we assume each microservice has an infinite capacity. Similarly for the service rate of a microservice, we assume that the service rate remains independent. Thus, even if the queue is very long, the service time remains identical. In case these assumptions

are too heavily violated, it is always possible to use a different approximation technique in this approach. In fact, we represent a microservice by the distribution characteristics of the arrival rate and service time. Changing the distribution seems straightforward, but the distribution for the merged microservice becomes non-trivial and requires different analysis based on the chosen distributions.

Another possibility is to create a (discrete event) simulation of the application. Based on the simulation, the required metrics can be approximated, such as the utilization and mean waiting time. However, simulations are typically computationally more expensive, resulting in an increased computation time of the genetic algorithm. Hence it is recommended to keep the approximation as fast as possible.

D. Fitness Objectives

Based on the fields of Queueing Theory and Software Module Clustering, several possible objectives were studied.

An important concept in queueing theory is the mean sojourn time, defined as total time a customer spends in the system, i.e. the waiting and service time combined. The mean sojourn time is directly related to the user perceived performance of a microservice system. This is an important factor in microservice architectures, as asynchronous messaging is used between microservices to propagate changes. Additionally, if the sojourn time becomes larger, a user is more likely to see an inconsistent state of the system, by viewing data from an internal feature instance that has not processed the latest change yet. Hence, it is also desired to keep the sojourn time as low as possible from a usability perspective. As the mean waiting and service time are part of the sojourn time, these objectives are not used individually.

Utilization, a measure of the used capacity, is another central concept in queueing theory. Unused capacity is basically wasted money for organizations, hence they aim to maximally use the available capacity. Typically a utilization between 60 and 80 percent is desired, as capacity is then efficiently used and there is always capacity to handle peaks in the workload. By combining features in a microservice, the workload handled by a single microservice increases, which increases the utilization of such a microservice. Hence this objective should be considered optimal when the utilization is between 60 and 80 percent.

In Software Module Clustering, several objectives are used to measure the fitness of a clustering of software modules [11][18]:

Maximize number of intra-edges Intra-edges are dependencies within a cluster. A high number of intra-edges indicates high cohesion.

Minimize number of inter-edges Inter-edges are dependencies between clusters. A low number of inter-edges results in low coupling.

Maximize cluster count To prevent a single huge cluster containing all modules.

Minimize single module clusters To prevent every module becoming its own cluster.

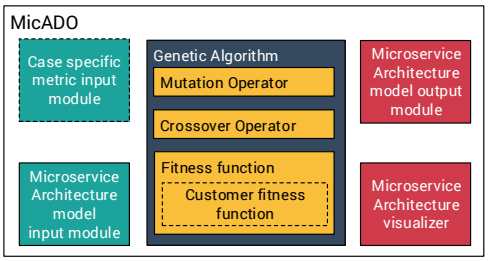


Fig. 4: MicADO components

In the case of a microservice architecture, inter-edges, i.e., edges between two microservices, do not exist, as these are resolved by adding internal feature instances to satisfy the feature dependency graph. However, adding internal feature instances results in data and code duplication. Which in turn requires an increase in communication, as each feature request needs to be propagated to more internal feature instances in different microservices. This results in increasing sojourn times, as microservices have more propagated messages in their queues. Furthermore duplication of features results in reduced maintainability. Therefore, we should minimize the number of internal feature instances.

As the latter two objectives, maximizing cluster count and minimizing single module clusters, are already encoded in the utilization and sojourn time, these were discarded in the fitness objectives.

E. MicADO

We created the open source MicADO: Microservice Architecture Deployment Optimizer [1] tool in which we implemented this approach. An overview of the components of this tool is depicted in Figure 4. Blocks with a dotted line indicate customer specific modules that can be overridden.

A technical representation of the described microservice architecture model and a workload model are the input of this tool. The workload model requires an application specific adapter that produces the expected workload model. After parsing these input models, they are passed to the genetic algorithm.

Based on the objectives of the company, our fitness function can be used, or a different fitness function can be implemented. Finally the 'Microservice Architecture model output module' outputs the best deployment suggested by the genetic algorithm.

Additionally, MicADO contains a web-based microservice architecture model viewer that visualizes the suggested microservice architecture model.

[1] www.architecturemining.org/tools/micado

V. Case study

A case study was performed to evaluate the approach. This section describes the case study context and the results.

A. Case Study Context

The case study was performed at AFAS. AFAS is a Dutch vendor of ERP software. The privately held company currently employs over 350 people and annually generates 100 million of revenue. AFAS currently delivers a fully integrated ERP suite which is used daily by more than 1.000.000 professional users of more than 10.000 customers.

The NEXT version of AFAS' ERP software is completely generated, cloud-based, and tailored for a particular enterprise, based on an ontological model of that enterprise. The ontological enterprise model will be expressive enough to fully describe the real-world enterprise of virtually any customer, and as well form the main foundation for generating an entire software suite on a cloud infrastructure platform of choice: AFAS NEXT is entirely platform- and database-independent. AFAS NEXT will enable rapid model-driven application development and will drastically increase customization flexibility for AFAS' partners and customers, based on a software generation platform that is future proof for any upcoming technologies.

B. The Architecture

Currently the generated architecture is an event driven microservice architecture using Command and Query Responsibility Segregation (CQRS) [8], and Event Sourcing [17] running on Microsoft Service Fabric at the back-end, and a HTML5 single page application as front-end. AFAS NEXT supports the generation of different feature groupings at the query side of its CQRS backend. The AFAS NEXT generation pipeline was modified to support our microservice architecture model as auxiliary input for the generation process. This microservice architecture model is used to distribute the projectors over the microservices. Because of the ability to modify the application generation process, it is possible to generate many different groupings of features. This makes AFAS NEXT a powerful environment for this research.

The applications ran on a five-node Service Fabric cluster, running on virtual machines. The databases of the microservices were stored on a dedicated database machine.

C. Microservice Architecture Model

The NEXT platform was used to create an application that resembles a web-shop, depicted in Figure 5. It contains a *customer* model element that enables visitors to create an account. A customer represents a person and consists of an email, password, default shipping address, and other personal details. Secondly the web-shop contains *products*, consisting of product information and several technical properties. These products can be reviewed by a customer, using the *review* event. A customer can create *orders*, consisting of one or more *order lines* containing an amount and a product. Furthermore

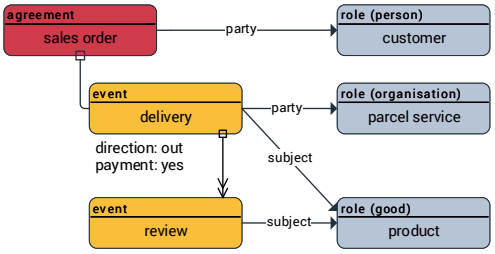


Fig. 5: Model of the created AFAS NEXT application

an order consists of a shipping address, by default the customers's default shipping address. Finally an order results in an *payment* and a *delivery*. A delivery is an event that results in goods leaving the organisation, requiring a payment in return of the other party, as depicted by the properties of delivery in Figure 5. A payment contains a dependency on the total price of an order. The delivery is performed by a *parcel service*, and uses the shipping address provided on the order.

These six model elements result in 27 features with a total of 238 properties and 72 dependency relations between features. The maximal microservice architecture is used as a baseline for the performance tests. The resulting microservice architecture consists of 25 microservices, 27 public feature instances and 55 internal feature instances, as shown in Figure 6(a). The number inside a feature indicates of which model element it originates. Features with a 1 or 2 originate from the sales order model element. Features 3 to 6 are created as result of the customer model element, and features 7 till 10 are the result of the Delivery element. The parcel service model element resulted in features 11 to 13. The review event resulted in feature 14, while the product role resulted in 15 and 16. Finally NEXT by default generates features 17 till 27, that were not used in this workload.

D. Workload

We created an artificial workload, since AFAS NEXT is still under development and not running in production. The workload we created consists of a typical scenario for a web shop. At first a user creates a shopping basket and adds several products to it. When the customer is done shopping, he pays for his order, and a delivery slip is created. Afterwards, some users submit a review of the product.

E. Test setup

The tooling by Guelen [6] is used to generate the workload. Unfortunately the tooling is currently not able to send a single request and wait till the event has been processed by all projectors on the query side. Since the steps in the workload require the system to be consistent after every step, this scenario had to be converted to a batch workload to circumvent this limitation. This resulted in a phase for every step described

in the workload above, in which all concurrent users perform that step. For example in the first phase all shopping carts are created, followed by a second phase in which all products are added to all shopping carts.

Since this workload is a burst process, which is not a poisson distribution, we were unable to use our queueing theory approximation and created a simulation.

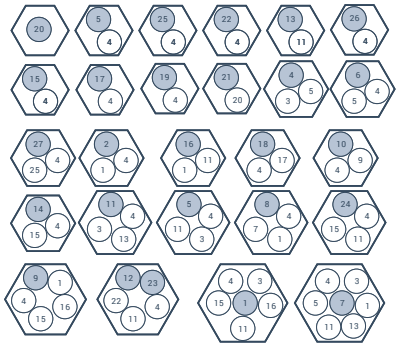
To evaluate MicADO, we created an AFAS specific metric input module that derives both the workload and the performance metrics from their Software Operation Data. Based on goals of AFAS, we decided to use two fitness objectives: mean time till consistency with a weight of one, and feature duplication with a weight of 0.2. Both objectives had to be minimized. Furthermore the simulation was integrated in the fitness function.

Before each performance test, base data is inserted in the system, such as accounts and products. In this phase, 251 countries, 250 users, 1760 products and four parcel services are created, that are used in the other phases of the performance test.

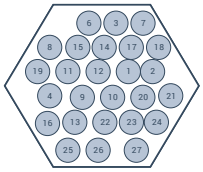
Two variants of the workload described in the previous subsection were used: the low and high variant. The low variant simulates traffic representing only a few users. This is done by running a single load generator thread that waits uniformly between 50 and 200 milliseconds between each request, i.e. the application has to handle between 5 and 20 requests per second. This tests is designed to determine a good clustering of features for small customers. Since the load on the system is low, the system should be immediately consistent, and no significant queueing of requests should occur.

The high variant simulates a busy day for the web-shop. This is done by using ten load generators with the same settings as in the low variant. As a result, the application has to handle between 50 and 200 requests per second. The workload should result in significant queueing occurring at most places in the application. As a result of this queueing, the system will require several minutes catch-up time to become consistent.

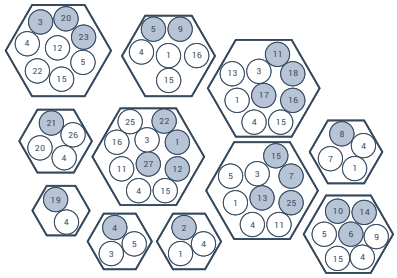
For both variants, five runs of the performance tests on the maximal microservice architecture depicted in Figure 6(a) were performed. Afterwards the run that is the closest to the mean of the five runs was selected. This run was used as input for MicADO, since the metric input module operates directly on the metrics. The microservice architecture model suggested by MicADO was used as input for the AFAS NEXT generator. The regenerated application was redeployed, and the performance tests were re-ran on this new architecture. The following section discuss the results for the low and high workload scenarios.



(a) Initial microservice architecture of the application



(b) Recommended architecture of the application for the low traffic workload



(c) Recommended architecture of the application for the high traffic workload

Fig. 6: Different microservice architectures for the feature model depicted in Figure 5

F. Results

1) Low Workload Scenario: The performance test results of the maximal deployment for this scenario are shown in the left part of Table IV. Every row in this table represents one of the phases of the workload. The time column denotes the total time in seconds it took the system handle this workload and become consistent. The avg. requests column denotes the mean number of requests the system handled per second. Note that a lower time is better, while a high number of avg. requests per seconds is better.

The metrics emitted during this test run were used as input for the MicADO tool. MicADO recommends to merge all features in a single microservice, resulting in a microservice containing 27 public feature instances with zero duplication, as shown in Figure 6(b). According to the simulation this should reduce the mean time till consistency from 183 to 62

milliseconds, while reducing the number of internal features instances from 55 to zero.

The results of the performance tests on the deployment based on the suggestion of MicADO are shown in the right part of Table IV. It should be noted that the time of all phases of the performance test were completed in less time than the initial microservice architecture. Secondly the mean number of requests per seconds is higher for every phase.

TABLE IV: Performance test results for the low workload variant, before and after optimization with MicADO and redeployment of the Microservice Architecture.

Test phase	**Initial MSA**		**Optimized MSA**	
(# requests)	Time (s)	Avg. requests/s	Time (s)	Avg. requests/s
Order (600)	131	4.56	104	5.75
Article (1200)	270	4.44	208	5.74
Payment (600)	132	4.53	104	5.53
Delivery (600)	126	4.74	108	5.53
Review (500)	108	4.61	89	5.58

2) High Workload Scenario: The results of the high workload scenario performance test on the maximal deployment are shown in the left part of Table V. This table has the same layout as Table IV, the table denoting the results for the low workload scenario.

Based on the metrics that were emitted during the run, MicADO recommends the deployment shown in Figure 6. This microservice architecture consists of eleven microservices, with a total of 39 duplicated internal feature instances. As can be seen in Figure 6, several microservices have been merged. According to the simulation performed by MicADO, the mean time till consistency increases from 1584 milliseconds to 1612 milliseconds. These numbers indicate that the full parallel processing capacity of the maximal deployment is fully used to handle this workload.

The application was again regenerated and redeployed based on the deployment suggested by MicADO. The results of the performance test performed on this architecture are shown in Table V. The results of these tests were close to the initial deployment, as shown in Table V, however with a much lower number of internal feature instances.

G. Case Study Evaluation

In case of the first workload scenario, the total time of the performance test is reduced with 20% and the throughput of the system increased with 23% on average. Hence MicADO was able to substantially improve the performance of the application. Since this scenario only puts a low workload on the system, a single microservice is able to process all requests without large waiting times. For this workload the overhead of processing every request multiple times by different microservices is larger than the benefits gained by the increased parallel processing.

TABLE V: Performance test results for the high traffic variant, before and after optimization with MicADO and redeployment of the Microservice Architecture.

Test phase	**Initial MSA**		**Optimized MSA**	
(# requests)	Time (s)	Avg. requests/s	Time (s)	Avg. requests/s
Order (6000)	130	45.83	121	49.30
Article (12000)	237	50.55	230	52.06
Payment (6000)	115	51.88	117	50.96
Delivery (6000)	118	50.60	116	51.43
Review (5000)	66	52.98	69	51.51

In the high workload scenario, the performance could not be improved substantially, but the second objective of the fitness function, the duplication of feature instances, could be reduced with 30%, from 55 to 39. As indicated by the simulation, the full parallel processing capacity of the application is required to handle the high workload. Combining several microservices that contain the same internal feature instances, resulted in an substantial decrease of the duplication, without a negative impact on the performance.

The two previous scenarios show that the different scenarios result in a totally different deployment, which both were able to substantially improve the overall fitness of the deployment for the defined fitness goals.

VI. Conclusions and Future Work

This paper contributes to the research on microservices in several ways. First, a formal notation to model microservice architectures is proposed. Based on this model, modification operators on microservices are defined. These operators, *move* and *merge*, move a feature from a microservice to another and merge two microservices respectively. Secondly, an approach is proposed to optimize the performance of a microservice architecture given its workload. This approach has two main components as input: a microservice architecture model and a workload with corresponding performance metrics. A genetic algorithm searches for deployments having a better performance for the provided workload. Since an actual execution of the workload on a deployment is computationally expensive, an approximation using queueing networks with multiple customer classes is proposed as fitness function. The third contribution is the creation of MicADO, an open source tool, in which we implemented the proposed approach, that can be used to easily implement the proposed approach in practice.

Finally a case study was performed to evaluate the proposed approach and MicADO. Results of the case study show that performance improvements up to 20% can be obtained in some cases. This underlines the importance of an approach that takes performance metrics into account when determining the size of microservices.

Based on this research, we see many opportunities for future work. Further research into the robustness of the genetic algorithm is required, such as robustness against small variations in workload, and reducing the effect of non-determinism in the algorithm. More case studies are essential to further optimize our approach, preferably with a real production workload, to confirm the outcome of our research.

There are many ways of defining optimality for microservices, performance and data-duplication are just two of a plethora of possibilities. Additional objectives, such as availability, security and maintainability require new research and

case studies. Furthermore, we believe that patterns in the workload can be exploited to improve the feature clustering.

Further research into efficient approximations of workloads is essential, since simulations are time consuming and are not flawless as shown in our case study. The approach in this paper has the potential to support self-optimizing architectures. We envision this by automating the feedback loop and generation of new microservice architectures based on the current and expected workload workload of the system.

Acknowledgments

The authors would like to thank Michiel Overeem, Dennis Schunselaar and Henk van der Schuur for the fruitful discussions and feedback.

This research was supported by the NWO AMUSE project (628.006.001): a collaboration between Vrije Universiteit Amsterdam, Utrecht University, and AFAS Software in the Netherlands. The NEXT Platform is developed and maintained by AFAS Software.

References

[1] W. M. P. van der Aalst, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[2] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Addison-Wesley, 2011.

[3] B. Chen and Z. M. Jiang, "Characterizing logging practices in java-based open source software projects a replication study in apache software foundation," *Empirical Software Engineering*, vol. 1–45, 2016.

[4] B. Davis, *Agile practices for waterfall projects: Shifting Processes for Competitive Advantage*. J. Ross Publishing, 2012.

[5] M. Fowler, "Microservices," 2014. [Online]. Available: http://martinfowler.com/articles/microservices.html

[6] J. P. J. Guelen, "Informed CQRS design with continuous performance testing," 2015.

[7] M. Harman, R. M. Hierons, and M. Proctor, "A new representation and crossover operator for search-based optimization of software modularization," in *GECCO'02*. Morgan Kaufmann, 2002, pp. 1351–1358.

[8] J. Kabbedijk, S. Jansen, and S. Brinkkemper, "A case study of the variability consequences of the cqrs pattern in online business software," in *EuroPLoP '12*. ACM Press, 2012, pp. 1–10.

[9] S. Lavenberg, *Computer performance modeling handbook*. Academic Press, 1983.

[10] R. Levy, J. Nagarajarao, G. Pacifici, A. Spreitzer, A. Tantawi, and A. Youssef, "Performance management for cluster based web services," in *IM 2003*, ser. IFIP Conference Proceedings, vol. 246. Kluwer, 2003, pp. 247–261.

[11] K. Mahdavi, M. Harman, and R. Hierons, "A multiple hill climbing approach to software module clustering," in *ICSM 2003*. IEEE Comp. Soc., 2003, pp. 315–324.

[12] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *IWPC'98*. IEEE Comp. Soc., 1998, pp. 45–52.

[13] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: a clustering tool for the recovery and maintenance of software system structures," in *ICSM'99*. IEEE Comp. Soc., 1999, pp. 50–59.

[14] J. Meier, S. Vasireddy, A. Babbar, R. Mariani, and A. Mackman, "Chapter 15 - Measuring .NET Application Performance," 2004. [Online]. Available: https://msdn.microsoft.com/en-us/library/ff647791.aspx

[15] B. Menouar, "Genetic algorithm encoding representations for graph partitioning problems," in *ICMWI*. IEEE Comp. Soc., 2010, pp. 288–291.

[16] D. Namiot and M. Sneps-Sneppe, "On Micro-services Architecture," *International Journal of Open Information Technologies*, vol. 2, no. 9, pp. 24–27, 2014.

[17] Michiel Overeem, M. Spoor, and S. Jansen, "The dark side of event sourcing: Managing data conversion," in *SANER 2017*. IEEE Comp. Soc., 2017, accepted.

[18] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Trans. on Softw. Eng.*, vol. 37, no. 2, pp. 264–282, 2011.

[19] C. Richardson, "Event-driven architecture," 2014. [Online]. Available: http://microservices.io/patterns/data/event-driven-architecture.html

[20] P. Schobbens, P. Heymans, and J.-C. Trigaux, "Feature diagrams: A survey and a formal semantics," in *RE 2006*. IEEE Comp. Soc., 2006, pp. 139–148.

[21] H. van der Schuur, "Process improvement through software operation knowledge," Ph.D. dissertation, Utrecht University, 2011.

[22] H. van der Schuur, S. Jansen, and S. Brinkkemper, "Reducing maintenance effort through software operation knowledge: An eclectic empirical evaluation," in *CSMR 2011*. IEEE Comp. Soc., 2011, pp. 201–210.

[23] J. Stenberg, "Martin Fowler on Characteristics of Microservices," 2014. [Online]. Available: http://www.infoq.com/news/2014/11/gotober-fowler-microservices

[24] Thoughtworks, "Microservices — Technology Radar — ThoughtWorks," 2014. [Online]. Available: https://www.thoughtworks.com/radar/techniques/microservices

[25] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An analytical model for multi-tier internet services and its applications," *SIGMETRICS Perform. Eval. Rev.*, vol. 33, no. 1, 2005.

[26] J. M. E. M. van der Werf and H. M. W. Verbeek, "Online compliance monitoring of service landscapes," in *BPM Workshops, Revised Papers*, ser. LNBIP, vol. 202. Springer, 2015, pp. 89–95.

[27] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," in *ASPLOS 2011*. ACM Press, 2011, pp. 3–14.