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 its performance and availability. 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 [23]. 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, 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 [22]. 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 two-pizza team (two pizzas are enough to feed the entire team) per service [5]. Another typical answer is that a microservice should do only one thing, which leaves room for interpretation.

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

Moving features to other, possibly 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 used frequently.

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 deployment model that describes the properties and dependencies of the features the architecture should implement, and 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.

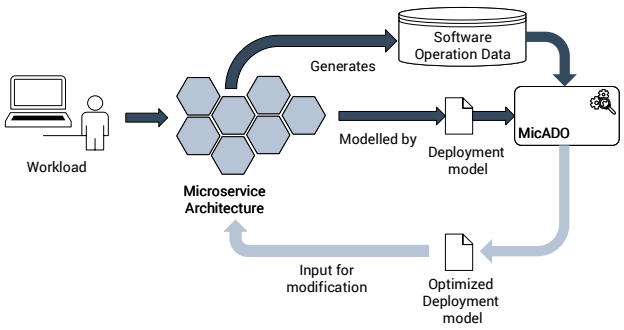


Fig. 1: Overall overview of the proposed approach.

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

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

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 different features.

A. Feature Model

A feature is a chunk of functionality that delivers business value [4]. It is represented by a set of unique properties together with a set of properties of other 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 favourite 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.

Notice that we allow properties to depend on properties within the same feature. An example feature model is depicted in Figure 2. This example consist of a subset of features typically found in a web shop: an invoice (A), an order (B), and customer feature (C). The invoice has a payment term P_1, an invoice date P_2, and an identifier P_3, based on both the order identifier and the customer identifier. The order has an identifier P_4, based on the name of the customer, a delivery address P_5, and order date P_6. Finally, Customer has an identifier P_7, a name, P_8, and address P_9. In our model, we thus represent feature A by $\{P_1, P_2, P_3\}$, and the set of all features F by $\{\{P_1, P_2, P_3\}, \{P_4, P_5, P_6\}, \{P_7, P_8, P_9\}\}$.

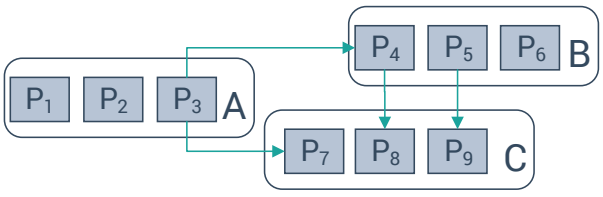


Fig. 2: Example feature model with three features: Invoice (A), Order (B) and Customer (C), each having three properties.

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. In this model, features publish events to which other features in other microservices can subscribe. These events allow the system to maintain data consistency across microservices without using distributed transactions [19]. The logic needed to handle these events is identical to the logic of the main feature, hence we consider this to be duplicated features. 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 one public feature instance and zero or more internal feature instances. An internal feature instance contains a non-empty subset of the properties of its 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* $\lambda : I \to 2^P$, 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 fulfill 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 the properties of its feature, i.e.
  $\forall i \in I : \exists f \in F : \lambda(i) \subseteq f$

- Each feature has a public instance that contains all its properties, 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 are three microservices, m_O, m_C and m_D. 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 $m_C = \{i_C\}$.

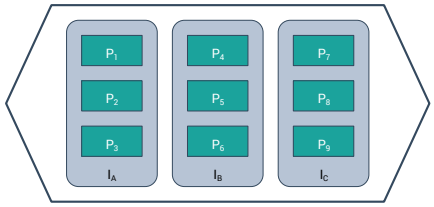
III. Measuring workload through Software Operation Data

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

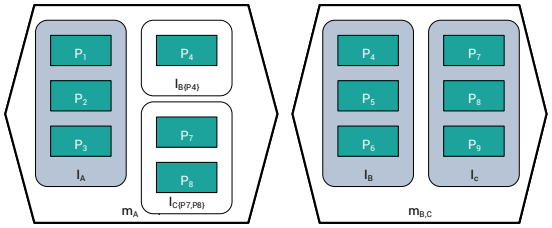
One way to obtain the workload of a deployed microservice architecture is by monitoring its operation. Monitoring the operation of a system is not new and the use of software operation data [21, 25] is widely used in software engineering practices [3], such as maintainability [21], problem diagnosis [26] and compliance [25]. In the remainder of this section, we utilize software operation data to obtain both the usage and the performance of a deployed microservice architecture.

A. Feature Usage

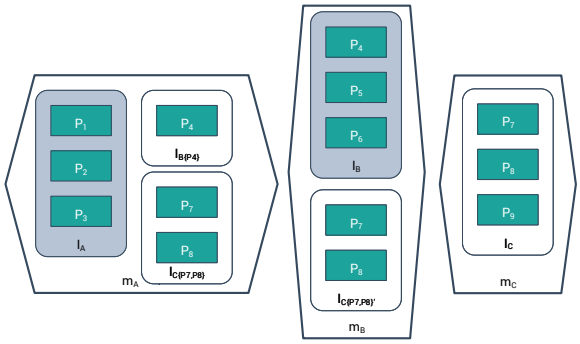
Feature usage over time provides valuable insight in frequent usage patterns, and therefore it is worth optimizing for. In the case of a microservice architecture, most communication is performed via lightweight mechanisms, such as the HTTP protocol, that support logging out of the box. In essence, a 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 logs, which contain information about which feature has been called, by whom and when, it is possible to derive the usage of features at the server. Process Mining [1], which analyses event logs to discover, monitor and improve real processes, allows us to analyse these access logs to derive relevant feature usage metrics.



(a) Minimal deployment



(b) Mixed deployment



(c) Maximal deployment

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

B. Performance Metrics

Feature usage is only one aspect of the 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. Metrics can be derived from software 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 individual features a microservice implements. Different levels of performance metrics can be identified [14]:

Application metrics Application metrics are metrics reported by the application itself.

Platform metrics Platform metrics are metrics reported by the framework or runtime of the application.

System metrics System metrics are reported by the operating system and/or hardware of the server.

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 supports structured metrics, to store relevant metadata as well to link these metrics 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, which 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 Praditwong et al. 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, the problem should be 'genetically' encodable, such that the genetic operators mutation and crossover are able to transform a 'chromosome', i.e. a possible solution, 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 a possible signal that the solution is sub-optimal for 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 are 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 [24].

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 processes 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:

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

In other words, 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 denotes 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 the 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. Common wisdom states that a utilization between 60 and 80 percent is desired, as capacity is then used efficiently 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 that 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.

In the case of an (event-driven) 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.

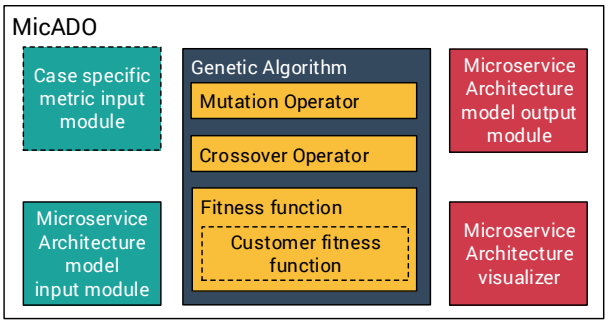


Fig. 4: MicADO components

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

To validate our approach, we created the open source tool MicADO: the Microservice Architecture Deployment Optimizer [1]. An overview of the components of MicADO 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.

V. Case Study

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

[1] www.architectureminingorg/tools/micado

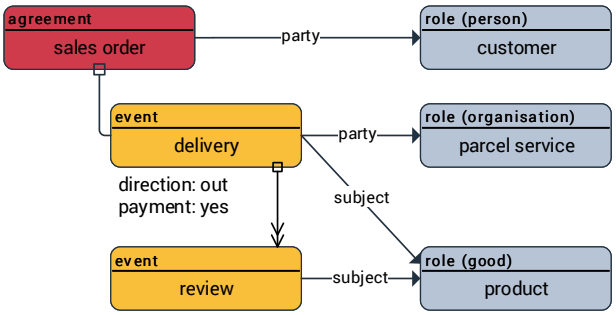


Fig. 5: Model of the created AFAS NEXT application

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 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 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 generates by default 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, since this tooling is integrated with AFAS NEXT. 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 therefore, we created a simulation model.

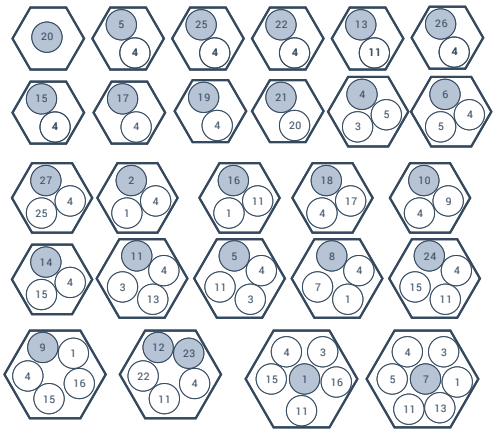
To evaluate MicADO, we created an AFAS specific metric input module that derives both the workload and the performance metrics from their software operation data. Based on goals of AFAS, we decided to use two fitness objectives: mean time till consistency with a weight of one, and feature duplication 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.

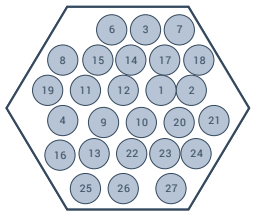
Two variants of the workload described in the previous subsection were used: the low and high variant. The low variant simulates traffic representing only a few users. This is done by running a single load generator thread that waits uniformly between 50 and 200 milliseconds between each request, i.e. 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.

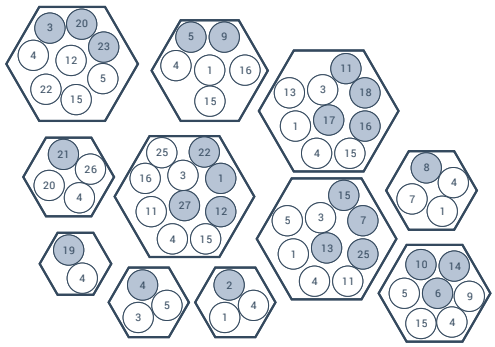
For both variants, five runs of the performance tests on the maximal microservice architecture depicted in Figure 6(a) were performed. Afterwards the run that is the closest to the mean of the five runs was selected. This run was used as input for MicADO, since the metric input module operates directly 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 discusses 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. Each 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, whereas a high number of average 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 scenarios show that the different scenarios result in totally different deployments, which both were able to substantially improve the overall fitness of the deployment for the defined fitness goals, given the respective workloads.

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. 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 in this case a performance improvement up to 20% was obtained. As the results stem from a single case study, we cannot generalize these results, but the case study shows 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, 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.

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

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

[24] 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.

[25] 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.

[26] 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.