Literature Review

Student
Maximilian Schiedermeier

Supervisors
Prof. **Bettina Kemme** - Distributed Information Systems Lab
Prof. **Jörg Kienzle** - Software Engineering Lab

Presented for the
Comprehensive Exam, Winter 2020

McGill
UNIVERSITY

School of Computer Science
January 10th, 2020

Models for Micro-Service Architectures

Abstract

Since a first mention in 2011, the Micro-Service Architectural style (MSA) has steadily gained in popularity. Notably in industrial contexts, MSA holds status of an established alternative to Service-Oriented Architectural (SOA) and monolithic styles.
The reason for this development are a variety of desirably software qualities that result from a clean implementation, adherent to the MSA style. Those are: faster development cycles, higher maintainability and better scaling. However, besides these advantages the MSA style also brings new challenges.
These challenges are inherent to the style and therefore appear in any MSA adherent project. At the same time many projects showcase recurring technical solutions for these challenges. This motivates a systematic reuse of tested and proven strategies.
In general, reuse is considered a good practice in software engineering. Yet it is unclear how MSA-specific solutions could be best modeled, to leverage reuse. Once integrated, alternative solution variants may furthermore result in different outcome qualities. Therefore, any model that supports reuse should also indicate the consequences of potential reuses.

My PhD focuses on the exploration of **models that leverage the quality of MSA applications**, as a contribution to resolve this challenge.

Although this constitutes an open challenge, the research question touches several well studied areas. In this literature review, I provide an overview of related contributions, to demonstrate familiarity with the broader chronological development of research around the above challenge.

1 Micro-Services and related Technologies

For most software domains, outstandingly successful projects feature similar design elements. Proven solutions and tested practices can be gathered and re-formulated as an architectural style. In return, the adherence to an architectural style can be beneficial for new software projects. A fitting style guides design decisions, reduces development costs and promotes desirable software qualities.
An established style for distributed software systems is the *Service Oriented Architecture* style. SOA modularizes software into functional units that in combination provide more complex operations to the end-user. While these individual services commonly expose their functionality via *Remote Procedure Call*[1] (RPC) interfaces, the entire communication is channelled through an application wide bus. SOA emphasizes many best practices that are

[1]RPC means that required functionality can is invoked, by delegating a task to a remote entity. At code level the invocation is comparable to an ordinary function call.

beneficial for reliable and maintainable distributed software systems, therefore it gained popularity for industrial projects.
However, over the recent years a new architectural style has emerged, and nowadays constitutes an established alternative to SOA. Notably for scenarios where scalability is a key factor, the *Micro-Service Architecture* style is increasingly preferred over SOA. A representative domain are e-commerce applications, where steady response times are commonly imposed by contract. One reason why respecting those constraints is challenging, is the burstiness of user generated load. MSA provides concepts to efficiently ensure responsiveness, while preserving general advantages of modular architectures.
The remainder of this chapter is structured as follows: In section 1.1 I present a set of software qualities that were not sufficiently answered by earlier styles, specifically SOA. Section 1.2 compiles MSA concepts and best-practices. Those are then compared to paradigms of the predecessor, SOA. This section also includes arguments to elaborate the purposefulness of distinctive characteristics for the satisfaction of initially targeted software qualities. Finally, in section 1.3 I present tools that leverage MSA's conceptual match on the initially targeted qualities with hands-on technologies.

1.1 MSA Origins and SOA Limitations

This section outlines how advancing standards for software qualities leveraged MSA as a new architectural style. Specifically, I focus on three fundamental software quality requirements: *Modularity*, *Scalability* and *Flexibility*. I illustrate how well SOA, the direct predecessor to MSA, supports these qualities. The following arguments are compiled from two contributions: *Dragoni et al.* [4] retrace the origins of MSA. This also covers a discussion on SOA. Furthermore *Rademacher et al.* [19] investigate the evolution from SOA to MSA from a modeling perspective.

1.1.1 SOA limits Modularity

Modularity is a metric that expresses to which extent a system is composed of coherent, loosely coupled modules. In practice the modularity of a software project is reflected at two levels: source-code and executables. The advantages of modularization at code level result from the fundamental principles of object-oriented programming [4, p5]. However, a modularization at code level does not imply separated executables. Modularized code may still be compiled into a single binary, also referred to as *monolith* [4, p1]. SOA as a style explicitly promotes modularization at code level and at executable level. The targeted unit of modularization are then the individual services. However, SOA does not provide clear guidelines on the granularity and boundaries of services, consequently they tend to orient on organization structures, e.g. departments, rather than functional coherence [4, p5]. As a consequence, SOA applications are often coarse grained and individual services become complex.
According to [19, p3], another design criterion of SOA, that hinders modularity, is to support a high level of interface abstraction. Incompatible services are connected through an enterprise bus that provides the required message transformations [19, p3]. The only way to avoid the bus as a bottleneck, is to unify the technologies of dependent services, to allow a direct

communication despite custom protocols and message formats. This induces a technological coupling between services and hinders modularity.

1.1.2 SOA limits Scalability

Scalability in the context of distributed applications is the ability to manage increasing load. A more formal definition and metrics are provided in the chapter 2. In contrast to monoliths, modular architectures enable a targeted replication of critical component instances. If under critical load, this allows an efficient allocation of additional resources, whereas monoliths require a full system replication. SOA systems typically provide one executable per service, and therefore, have a better scalability than monoliths [19, p5]. Targeted replication of critical system components is restricted by low service granularity. As SOA applications are typically coarse-grained, i.e. consisting of relatively few, complex services, targeted replication of critical components is not straightforward. This limits scalability for resources are not optimally allocated.

1.1.3 SOA limits Flexibility

Requirements on a software product constantly change [7, p8]. A flexible architecture allows the fast adapting to changing requirements. Modularity plays an integral role to flexibility, because modules encapsulate functionality and complexity, and therefore, facilitate code changes. If a changing requirement can be delimited to a specific module, this supports flexibility. SOA supports modularity with service-oriented code-bases that can be updated individually. But even though this eases updates on code level, the modifications are meaningless to the end-user, until the updated code is compiled and deployed on the production system. SOA does not foresee module-oriented continuous integration [19, p4]. This means that it does not include elaborate concepts for fast updates of individual services. In practice this means that code changes remain without effect until the next full release.
In summary, SOA provides essential concepts that to some extent support *modularity, scalability* and *flexibility* as desirable qualities. Notably for distributed applications, SOA induces many advantages, that are absent in monolithic architectures. However there are limitations.

1.2 MSA Concepts and Distinction to SOA

The MSA style provides specific concepts that address the challenges discussed for SOA. The style does not constitute a complete revocation of SOA characteristics. It rather amends the style at critical points. It is therefore sometimes referred to as *the second iteration on the concept of SOA* [4, p5] or *fine grained SOA* [4, p6]. However, it is hard do draw a clear line. Not all MSA-adherent implementations necessarily showcase all indicators. At the same time, SOA adherent implementations may feature some MSA typical indicators. In addition to that, the set of characteristics varies, depending on the reference consulted.
Consequently, in certain cases a classification into binary categories is inadequate. However, the more characteristics an implementation exposes, the clearer it can be characterized as MSA adherent.
In this section, I present a set of characteristics that is commonly considered MSA typical.

It is extracted from the following references: *Kakivaya et al.* [14] describe "Service Fabric", a platform designed to host Micro-Service applications. Their contribution includes an accurate recapitulation of common MSA characteristics. *Jaramillo et al.* [12] summarize how virtualization and build & deployment technologies leverage the advantages of MSA. Finally much insight comes from *Eismann et al.* [24], which published an open source web-shop named *Tea-Store*, intentionally designed to showcase maximum adherence to MSA. They then published detailed technical insight and documentation for this MSA application. One of the goals of Tea-Store is, to improve experiment comparability, by establishing a reference implementation for MSA related case studies.

1.2.1 Common MSA Characteristics as Indicators

1. Common to all specifications are polyglot services. One of the most outstanding characteristics of MSA applications is, that each individual service can be developed in the language that fits best, regardless of the remaining services [24, p3] [12, p2] [14, p1].

2. Compared to SOA, MSA services emphasize higher cohesion, resulting in finer grained services. Selecting the services' granularity is a trade-off between performance and modularity [19] [22, c26]. Smaller units improve modularity and more targeted replications, but result in increased traffic, as smaller services require more communication to fulfil a meaningful task. Regarding the services' scope, there is a second evolution: SOA tends to define services around organization structures, whereas MSA targets business drivers. An example would be an e-commerce platform, where services represent product categories instead of sale-oriented services like a *shopping-cart*. This disruption from organization structures also affects the composition of teams. MSA advocates service-centrist developer teams, that showcase higher competence heterogeneity [12, p4].

3. The aforementioned team composition also plays a role regarding *Continuous Integration* (CI) and *Continuous Development* (CD) [4, p3]. Purpose of those are fast releases to quickly respond to changing requirements. MSA answers this by assuming service deployment as a responsibility of the team who develops a given service. Fast updates are further supported by container-based virtualization [14]. This also enables automated build and deployment pipelines [12, p4].

4. In MSA, services communicate over standard lightweight protocols. The de-facto standard are request-reply protocols, with RESTful service interfaces [24, p4]. The internal service structures and dependencies are hidden to the end user, setting a single API gateway as unique application access point [19].

5. Finally, MSA applications rely on *choreography* as default delegation strategy, whereas SOA uses *orchestration* [4, p7-p8]. The difference is that orchestration implies the existence of a central entity to determine which target entity will be a service representative. Choreography on the other hand, grants autonomy to the individual services. When a remote service has to be invoked, the calling service decides which of the available target instances will be called [19, p3].

1.2.2 A-Posteriori Motivation of MSA Characteristics

This section explains how the previously presented MSA characteristics support desirable software qualities.
As MSA is supposed to resolve SOA's initially presented limitations on *modularity, scalability* and *flexibility*, the newly introduced concepts must support these qualities. The following enumeration illustrates this for the aforementioned MSA characteristics:

1. Polyglotism improves *modularity*, as it reduces the coupling between services.

2. Fine grained services allow a targeted replication of bottleneck services. This allows a more efficient allocation of resources. Ultimately, it enables a better *scaling* of the overall system, when put under load. More different services furthermore imply a higher *modularity*.

3. CI and CD are strategies that target a fast propagation of code-changes into the production system. They render a project more *flexible*.

4. Standardized inter-service protocols and interfaces remove the need for an enterprise-bus, since there are no syntactic gaps to bridge [19, p3]. Not having to pass communication through a specific channel improves *scalability*. Furthermore standard protocols advocate polyglotism and therefore support *modularity*.

5. Choreography has a positive effect on *scalability*. Individual services can dynamically select the entity with best response time, for any given dependency.

Support of *modularity*, *scalability* and *flexibiliy* are essential prerequisites in the context of e-commerce applications. Those must provide stable response times, even if exposed to user load with high burstiness. Despite system complexity, a possibility for fast functional adaptations is demanded. Therefore MSA centrist case studies, such as the Tea-Store [24] elicit this domain [12].

1.3 Technologies to Gain MSA Characteristics

The previous sections presented MSA characteristics and illustrated how they support desirable software qualities. In this section, I present technologies that induce the above characteristics.
The list is compiled from previously introduced references, with one addition: *Roy Fielding et al.* [6] recapitulate key concepts of a RESTful interface design.

1. Polyglotism results from the multitude of programming languages that support standard network protocols. *Dragoni et al.* state that in an MSA context mostly object oriented or functional languages are used in practice [4, p6].

2. Fine service granularity does not require a specific technology. However, an accurate understanding of the targeted business drivers is required.

3. CI and CD heavily depend on automated build and deployment pipelines. *Maven* ensures an accurate configuration of the build process. This allows the direct generation of deployable containers from source code. *Docker* enables a parallel deployment of containers on a shared host kernel. Notably both tools can be automated with *bash* commands [14] [12].

4. REST is considered the de-facto standard for MSA [24, p4]. REST stands for a resource oriented representation of service functionality. This allows the definition of compact APIs with standard CRUD[2] methods. REST in principle is agnostic to the application layer protocol used. However, most implementations decide on HTTP, due its wide support by programming languages [6] [12]. MSA also involves publish-subscribe protocols in exceptional cases, notably if asynchronous one-to-many communication is required [19].

5. Choreography grants individual services autonomy on which target entity to invoke in support of own functionality. Selfish agents may select the target entity with best response time, while altruistic agents can spread the load. However, this implies the existence of a dynamic list of currently available instances. This can be provided by default utility services, such as registries that run a constant service discovery. Cloud platforms like Microsoft's *service fabric* and container-orchestration systems such as *Kubernetes* provide such utility services out of the box [19] [14].

1.4 Recap

MSA emphasizes specific software qualities better than its predecessor, SOA. However, the improvement is reached at a higher technological debt. There is no clear boundary between MSA and SOA. Furthermore the decision which style fits best is context dependent. MSA is a reasonable candidate whenever advanced demands for *modularity*, *scalability* and *flexibility* are present. This is particularly the case in the context of e-commerce applications.

2 Model-Based Performance Predictions for MSA

This chapter focuses on performance evaluations and predictions for MSA adherent applications. It is structured as follows:
I first present current strategies to quantify performance as a software quality and argue why especially response-times are an important metric in the context of MSA. It follows a brief motivation for performance evaluations and predictions. Finally, I present different prediction approaches and illustrate why it is challenging to obtain predictions that are fast and accurate at once.

2.1 Quantifying Performance

A meaningful evaluation of software qualities requires generic and universal metrics. However it depends on the quality, whether such an expressive metric exist. *Performance* as a

[2]CRUD: *Create, Read, Update, Delete*

quality is commonly quantified through *reponse time* [13, p1] and *throughput* [2, p2]. Both metrics leave little room for interpretation and therefore ease comparability. Another example for an easily quantifiable quality is *energy efficiency* [24].
Other qualities, like *resilience* are harder to quantify. An example where this attempt is made is the work of *Heorhiadi et al* [10]. The authors, propose a framework to quantify resilience, by performing a systematic verification of functional tests, while simulating network and deployment errors. The success rate of functional tests is then interpreted as the tested application's resilience. While this approach defines a metric, the outcome is not as generic or comparable as desirable. The obtained value depends on the applied tests in combination with the simulated errors. Their work illustrates how hard it is to accurately express a system's degree of resilience.
Other engineering-related qualities like *modularity* and *flexibility* are even harder to quantify. In chapter 3 I briefly illustrate how certain modeling strategies handle these qualities to establish a certain level of comparability.

2.1.1 Response Time as an MSA Key Metric

Not only is performance as a quality straightforward to measure, it is also one of the most relevant qualities for evaluating MSA adherent software. The MSA style emerged as a reaction to tighter demands on response times. In coherence to the MSA typical request-reply communication, so called *Service Level Objectives* (SLOs)[3] define the maximum delay between a request entering the system and a corresponding reply [13, p1]. These SLOs are contracted constraints. Exceeding the allowed response time is considered a violation of the contractually guaranteed software quality. Performance evaluations for MSA systems therefore mainly focus on response times.
The customer acts in the role of an external evaluator. Her interest lies entirely in the response time characteristics of the overall system. The service operator however has a motivation to also understand the performance of individual services. *Podolskiy et al.* [13] argue that SLOs should be answered in an economic way. As the allocation of resources is costly, they suggest to aspire deployments that just suffice for validating SLOs, while minimizing the hardware allocation. An important observation is that an incoming request can internally be handled by an interplay of many services. An efficient reduction of the externally measurable response time can be achieved by a targeted replication of only the bottlenecks. However, a targeted replication requires accurate measurements of individual service response times. *Podolskiy et al* [13] propose a framework that can determine such performance characteristics of internal services. Their approach runs in two stages: First, communication, resulting from external stimulus, is probed and recorded. Then, internal services are sandboxed to measure their individual response times for replayed input messages. If a sandboxed service attempts to call its dependencies, those queries are instantly answered with pre-recorded replies. The authors claim that the measured response times provide realistic service profiles.

[3]Sometimes also referred to as *Service Level Agreements*, SLAs [2], [18].

2.1.2 Need for Predictions

SLOs constitute non-negotiable performance requirements. At the same time the cost of hardware allocation encourages a fulfilment of response time limits with efficient allocations. While profilers provide insight on response time characteristics of individual services, they can not produce or analyse fictitious deployments.
It is practically unfeasible to first implement and then test the variety of all potential deployment configurations. A common approach are therefore performance estimations of potential configurations. *Correia et al.* [2] propose an adaptation of queuing models specifically for MSA applications. In their approach, they describe the performance of services under load, also referred to as *scaling*, through a stochastic model. When parametrized with an expected peak load, they can then estimate the response time of deployment configurations and elicit the most efficient among potential allocations.
The approach suggested by *Correia et al.* [2] is just one out of many. In the next section, I present further strategies and illustrate how the required input models differ. Afterwards, I discuss advantages and challenges of different strategies as well as their primary applications.

2.2 Advanced Response Time Predictions

Performance predictions require a specification of the MSA deployment in question. This is provided in form of a model. Different prediction methods require different models, have individual advantages and limitations, and target different performance aspects.
Eismann et al. [5] target a combination of existing approaches that unites the individual advantages. They first describe two current opposed methods, and then construct a hybrid approach. Specifically, the authors consider the following approaches:

- Event-based simulations are predictions that work on white-box MSA models. These models accurately define an application's components, deployment, internal dependencies as well as the available resources. An example is the *Palladio Component Model* (PCM) [5]. Scaling of a deployment is then estimated by simulating how inbound queries are processed by the defined resources. The simulation only depends on the provided model, therefore this approach can evaluate previously unseen deployment configurations. However, the high level of model details renders this method resource intense and time consuming [5]. This restricts the applicability of the approach, as it limits the amount of practically explorable configurations.

- Benchmark data of an existing MSA application can be used as training data for machine learning. System-scoped benchmark data is agnostic to the internal MSA structure or performance of individual services, but still allows predictions on the overall system performance. Once the model is trained, it therefore allows fast predictions. On the downside, machine learning predictions require representative training data. It is therefore not possible to evaluate the performance of fictitious deployment configurations, due to the lack of benchmark data that could be used for training [5].

Taking this as a starting point, *Eismann et al.* [5] propose a hybrid approach. They suggest to use machine learning to predict the performance of individual services, but retain the remaining MSA internal descriptions from the whitebox model. That is to say the macroscopic structure is modeled accurately, while the computational debt of individually simulated services is removed. Notably, service-scoped benchmark data is deployment independent and allows predictions for yet non-existent configurations. The authors claim that this hybrid approach allows to deal with larger and more detailed systems while retaining the prediction accuracy, within the same amount of simulation time [5, p9].

2.3 Online Predictions

Another consideration is for which use the predictions are generated. One strategy is to search for a deployment configuration, based on contractual response time constraints and an expected peak load. This means that predictions are made before the application is in service. These predictions are then called *offline performance predictions* [18].
However, often real-world system load is bursty. This notably holds for the MSA typical e-commerce context. A static deployment, based on only peak performance considerations is on average wasteful with resources. Most of the time there are more resources allocated than required to satisfy the contractual response-time constraints. A more economic approach is a dynamic allocation of resources, in correlation to the current load. An application may therefore itself explore more economic deployment alternatives at runtime, based on the momentary load. These constant background evaluations of potential alternatives are then called *online performance predictions* [18].
A representative online performance prediction tool is presented and evaluated by *Huber et al.* [18]. In their approach, a control loop constantly monitors response times and searches SLA threats as well as wasteful resource allocation [18, p4]. If detected, a model representing the current configuration is mutated to a variety of deployment alternatives. Their tool then runs performance predictions for all variants and elicits the most economic one. This process also takes a predefined set of constraints and potential insecurities into account. Finally the actual service deployment is adapted accordingly.
In their case studies, the authors were able to reduce resource allocation by 45 percent, compared to a static deployment [18, p17].

2.4 Recap

In the context of MSA, performance is measured by response-time under varying load. MSA applications have tight response time constraints. Operators aspire to meet these constraints with an efficient deployment that minimizes the allocated resources. Performance predictions are an essential tool to estimate the scalability of alternative deployments. Advanced systems constantly try to optimize their deployment with online performance predictions, to adapt to the current load.

3 Model Driven Software Engineering and Reuse

In this chapter I present key concepts of *Model Driven Software Engineering* (MDSE) and current model based approaches that target the systematic reuse of software solutions.

3.1 Models

Models are abstracted descriptions that intentionally restrict the represented matter to a certain viewpoint. Applied on complex systems, these abstractions can be beneficial for a better understanding, or enable further investigations on system qualities [9, p1]. In the previous chapter we saw how models of a system's deployment allow estimations on performance.

3.1.1 Modeling Languages and DSLs

What can be expressed or inferred by a model depends on the modeling language used. The elements of a model, also referred to as *concepts*, are defined by the modeling language's syntax. Furthermore models can be visualized in diagrams. However, these diagrams may only display a subset of model information [1].
In computer science a popular modeling language is the *Unified Modeling Language* (UML). UML was designed for a broad use and supports a variety of diagram types, e.g. *Class-diagrams* and *Sequence-diagrams*. These diagrams are likewise applicable for a variety of computer science domains. However, languages can also be tailored for a specific domain. Such languages are called *Domain Specific Languages* (DSL) [3, p1]. A prominent DSL example is *HTML*, as a language to model the layout of web-pages.

3.1.2 Meta-Models

Previously I mentioned that model concepts are defined by a corresponding modeling language. However, a modeling languages can itself be described through a model. Such a model is then called a *meta-model*. A meta-model covers not only concepts, but also their relations and this way defines an abstract syntax [3, p3].
In summary, a language's meta-model can be seen as a specification, that all adherent models must comply to. If a model uses a concept or relation not defined in the meta-model, it is not a valid model of the corresponding modeling language.

3.1.3 Meta-Model Hierarchies

Meta-models provide an abstract syntax definition for models. However, meta-models are themselves models and require a syntax definition. This again can be achieved with an additional model at a higher level of abstraction. *Meta-meta-models* define the abstract syntax of meta-models.
An example for the resulting layered hierarchy is the *Meta Object Facility* (MOF), specified by the *Object Management Group* [8]. Initially, there is only the actual object to be modeled, the *User Object*. Next comes the *User Model* as an abstraction of the user object. This first level of abstraction is referred to as M1. Level M2 is the UML meta-model. It defines the concepts and references used in M1 models. Finally at level M3 is the MOF provided meta-meta model [8, p6].
An important characteristic of the MOF's meta-meta-model is *self-containment*. This means the abstract syntax of the MOF meta-meta-model can be fully modeled with the own concepts. Therefore, no further levels of abstraction are required.

Also, since the UML meta-model's abstract syntax is described by the MOF meta-meta model [23, p340], UML is called a MOF compliant language [11, p2].

3.2 Modeling for Reuse

As mentioned in the beginning of this chapter, models are often created with a specific purpose in mind. For the remainder of this chapter, the focus lies on models that support the reuse of software artefacts.
Prior to technical considerations on how to best achieve this, I summarize the general importance of reuse for software engineering.

3.2.1 Motivation

In general a systematic reuse of existing code is beneficial for multiple reasons: The reuse of software artifacts reduces development time [7, p23]. If the reused code is provided through a tested and proven library or framework, this furthermore promotes the absence of bugs. A re-implementation of existing functionality on the other hand can introduce new programming errors.
Furthermore certain software projects inherently depend on systematic reuse. Software Product Lines (SPLs) generate customized variants of software. To generate those variants, shared commonalities of the SPL codebase are systematically reused.
Reuse as a concept is a powerful tool to improve the quality of software. However, there are different attitudes towards a meaningful granularity for the unit of reuse. Those are presented in more detail throughout the next sections. An interesting initial observation is, that they altogether stipulate a modular character for the matter of reuse. That is to say, the reused unit is meant to promote internal cohesion and low coupling to its environment [16, p1] [11, p1] [9, p4].
The following sections illustrate modeling techniques at two representative module granularities.

3.2.2 Component Based Software Engineering

Reussner et al. define a software component as "*[...] a contractually specified building block for software, which can be composed, deployed, and adapted without understanding its internals*" [21, p47]. Handling software components as reusable building blocks forms the field of *Component-based software engineering* (CBSE). *Kienzle et al.* define CBSE as "*[...] a reuse-promoting way of developing software that is based on defining, implementing and deploying loosely coupled, independent components*" [16, p1].
Kienzle et al. furthermore perceive a component as "*[...] a modular and cohesive unit [...]*" that "*[...] provides clearly defined functionality, encapsulated behind provided interfaces*" [16, p1]. This definition demonstrates the purposefulness of CBSE for an MSA context. The component definition matches the characteristics of MSA services, wired e.g. via RESTful communication.
In coherence to above CBSE definition, the corresponding models target on common CBSE activities. This is demonstrated by the *Palladio* suite, a software that provides modeling

languages and tools to define components, their dependencies and interactions, allocatable hardware resources and component deployments. While Palladio even supports using these models for performance predictions, the key interest lies in a systematic reuse of components. An essential concept within Palladio is a pre-filled *component repository* that gathers components, as they are crafted. Components of the repository can then be reused in future projects [21].
Although MSA services fulfil the criteria of CBSE components, an off-the-shelf reuse of services is arguable. There is only a limited amount of generic services that could be reused across various applications. Examples are: *loggers*, *service registries* or an *authentication* service. Nonetheless there is a variety of MSA related technologies that crosscut individual services.
The next section focuses on concern-oriented reuse, a related paradigm that changes granularity of the matter of reuse from components, down to individual technical solutions.

3.3 Concern-Oriented Reuse

According to *Kienzle et al.*, concerns as a unit of reuse *[...] provide a variety of reusable solutions for recurring software development issues* [16, p2]. Alike components, these solutions can be collected in a *reuse library* [16, p2]. Also the functional alternatives of a concern are assumed to be offered through *clear interfaces* [16, p2]. An example that illustrates high versatility, is an *Association concern*, that compiles various generic datastructure solutions, like *Lists, Sets, Trees, etc.* that altogether link data [17, p3]. This also illustrates the cross-cutting character of concerns. The matter of reuse is generic enough to reappear at many different parts of an application. Applied to an MSA context this would mean that many individual services, respectively components, internally showcase identical concerns.

3.3.1 MSA relevant Concerns

Concerns gather technical solutions. The composition of a concern library is therefore domain dependent. In an MSA context, it should primarily offer solutions for MSA related software development issues. *Zündorf et al.* [20] provide a collection of recurring MSA-specific solutions. The authors recognize that the high technological heterogeneity of MSA adherent applications justifies tools to support explicit technical decisions. In their work they first analysed a real-world MSA application for *technological variation points* (TVP). TVPs are considered "*[...] places in an MSA based software-sytem at which technology heterogeneity is accepted by intent*" [20, p2]. *Zündorf et al.* further argue that TVP variants can be categorized into cross-cutting concerns. That is to say, recurring solutions can be categorized by the purpose they serve. The main identified categories are *Deployment, Data Format, Infrastructure Technology, Programming Language* and *Protocol.* The provided outcome is a good overview of MSA related practices, that furthermore confirms the list of technologies presented in section 1.3.
In addition to this extraction, the authors suggest a meta-model that specifically targets above MSA-concerns.

3.3.2 Feature- and Goal Models

An actual reuse of concerns from a provided concern library requires dedicated models. The purpose of such models is an easy selection and configuration of relevant concerns, for later integration into an application. An illustration of variants in an MSA context are optional communication features, such as an increased request timeout, or a certain encryption standard to be applied.
According to *Mussbacher et al.*, feature models are the "*[...] de facto standard for variability modelling*" [17, p2]. A feature model can be visualized as a tree, where individual branches represent concern features. The user activates or deactivates individual features. Advanced feature models can furthermore contain *composition rules*, that serve as constraints for valid combinations [15, p65].
Horcas et al. suggest a further categorization of concerns by *functional quality attributes* (FQAs). FQAs are considered software qualities, that are primarily influenced by concern configurations. Provided examples are *security, usability* and *error handling* [11, p1]. An organisation into FQAs allows a goal-oriented analysis of selected features [11, p21]. Direct feedback on the impact of a concern and feature selection on resulting software qualities can be produced through *impact models* [17, p4]. Such an impact could for instance quantify how different encryption standards influence the overall application security.
Horcas et al. [11] suggest to use a MOF compliant language to express the base architecture, but to maintain a conceptual separation from FQA oriented feature models. They argue, that this allows "*[...] software architects to focus only on application functionality*" [11, p1]. Selected concerns and features are then retroactively injected into the corresponding base application.
The integration of concern-derived code at designated positions of a base context is called *weaving* [20, p3]. *Horcas et al.* [11] present several weave algorithms, also referred to as *weave patterns* and conduct a case study to evaluate the automated injection of selected concerns into an e-voting test application.

3.3.3 Concern Hierarchies and Delaying of Decisions

The *Concern Oriented REuse* (CORE) paradigm, suggested by *Kienzle et al.* [16] [17], unites the aforementioned concepts of feature models, impact models and weaving strategies. CORE offers three interfaces to streamline the stages that an application developer commonly takes to integrate prepared concern solutions. Those are the *Variation interface*, the *Customization interface* and the *Usage interface* (VCU). The *Variation* interface allows the selection of desired concern strategies though a feature model and impact model. The latter allows a tradeoff analysis. The *Customization* interface further adapts the generic outcome model variant to the applicative context. An example is the substitution of generic types by specific types. Finally the *Usage* interface targets interactions with the concern's functionality.
CORE also considers a hierarchical structuring of concerns. This emphasizes reuse, because it allows a concern to internally make use of other concerns. However, a layered arrangement also introduces new challenges. General versatility of a reusable unit contradicts assumptions on the applicative-context [17, p1]. On behalf of concerns at a higher hierarchical

level, this affects the configuration of internally used concerns. If the configuration of internal concerns is made prematurely, this can drastically reduce the versatility of higher level concerns. Alternatively generic applicability can be preserved by re-exposing all internal variability through the encompassing concern. The latter is commonly not desirable, because the combinatoric debt easily leads to overly complex interfaces.
CORE sidesteps the aforementioned challenge with a dedicated notion for *delayed decision making*. When selecting the features of a concern, delayed decisions are reflected in a third option: "*deferred*". This option complements the standard *selected / de-selected* feature options [17, p3]. If deferred, this explicitly means that the decision on an inclusion of the corresponding feature is delayed, until a meaningful decision is possible. A significant advantage of this approach is, that the re-exposed variability can still be reduced in hindsight. This way decision still reduce complexity by further restraining exposed features.
In addition to intentional uncertainties in feature models, *Kienzle et at.* provide the required algorithm to integrate delayed decisions into existing models though all three VCU interfaces, when they eventually arise [17, p2].

3.4 Reusing Meta-Models

Component- and concern oriented-models target units of reuse at different granularities. Yet both approaches have exclusive advantages: Models for component-oriented reuse can provide information on macroscopic structures and this way allow a corresponding quality analysis, such as performance predictions. Models for concerns, target a broad reuse of cross-cutting solutions. Concerns can be mapped on FQAs, and corresponding feature models therefore enable automated quality injections [11, p1].
Kienzle et al. [16] discuss the feasibility of hybrid models to combine the advantages of both model granularities. The authors investigate a potential integration of CBSE and CORE concepts, specifically through a partial fusion of the CORE and PCM meta-models [16, p5].
A fusion of meta-models, as proposed in above contribution, constitutes a general challenge. Although, meta-models are an essential tool to leverage software modularity, meta-models themselves are commonly not designed to the same standard. A systematic reuse of meta-models, or parts of meta-models, is hindered by their non-modular design [9, p2]. *Heinrich et al.* [9] therefore suggest a modular meta-model reference architecture to better support their reuse. The authors demonstrate the viability of their architecture, by using it as a guide for the modularization of an existing meta-model. Additionally they do the same for the creation of a modular meta-model from scratch [9, p3].
A systematic reuse of meta-models is not only relevant to bridge the gap between different model granularities. Reusable meta-models also find application in the context of DSLs. As the abstract syntax of DSLs can be defined through a meta-model, changes in a DSL specification are reflected in the corresponding meta-model as well [3, p3]. *Degeule et al.* [3] argue how modular meta-models can ease adaptations in fast-evolving DSL specifications and likewise support the design of new DSLs. Furthermore, *Degeule et al.* perform a internet-of-things related case study to demonstrate how modular meta-languages enable a convenient production of new DSLs, out of existing languages.

3.5 Recap

A systematic reuse of software artifacts is an essential concept in software engineering, as it decreases costs and supports software quality. An essential facet of model-driven development targets reuse. However different modeling approaches showcase different reuse granularities. Component-oriented models allow architecture-related evaluations, such as performance predictions. Concern-oriented models rather focus on feature models and automated integration of FQAs. Advanced concepts such as the hierarchical organization of concerns and delayed decision making can further leverage reuse. Hybrid models that target multiple levels of reuse are conceptually feasible. Modular meta-models can support a faster design of new elaborate modeling languages.

Outlook

This literature review demonstrates that research about micro-service architectures touches multiple disciplines. Although the MSA style brings the potential to leverage modularity, scalability and flexibility of distributed software systems, these advantages can only be gained if meaningful design choices are made throughout the entire engineering and deployment process. Notably model-based performance predictions and model based reuse are key concepts for this challenge. Yet, as the MSA style is still relatively young, many open questions remain to be answered, until elaborate tools provide meaningful assistance and thorough guidance. I anticipate this PhD as an exiting opportunity to contribute to this task.

References

[1] IBM Knowledge center. 2019. UML models and diagrams. `https://www.ibm.com/support/knowledgecenter/es/SS8PJ7_9.6.1/com.ibm.xtools.modeler.doc/topics/c_models_and_diagrams.html`

[2] Jaime Correia, Fábio Ribeiro, Ricardo Filipe, Filipe Arauio, and Jorge Cardoso. 2018. Response Time Characterization of Microservice-Based Systems. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE, 1–5.

[3] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: A meta-language for modular and reusable development of dsls. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 25–36.

[4] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. In *Present and ulterior software engineering*. Springer, 195–216.

[5] Simon Eismann, Johannes Grohmann, Jürgen Walter, Jóakim Von Kistowski, and Samuel Kounev. 2019. Integrating Statistical Response Time Models in Architectural Performance Models. In *2019 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 71–80.

[6] Roy T Fielding, Richard N Taylor, Justin R Erenkrantz, Michael M Gorlick, Jim Whitehead, Rohit Khare, and Peyman Oreizy. 2017. Reflections on the REST architectural style and principled design of the modern web architecture (impact paper award). In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 4–14.

[7] Eric & Elisabeth Freeman. 2004. *Head First Design Patterns, 1st Edition.* O'Reilly.

[8] Object Management Group. 2019. MOF 2.5.1 Specification. `https://www.omg.org/spec/MOF/2.5.1/PDF`

[9] Robert Heinrich, Misha Strittmatter, and Ralf H Reussner. 2019. A Layered Reference Architecture for Metamodels to Tailor Quality Modeling and Analysis. *IEEE Transactions on Software Engineering* (2019).

[10] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K Reiter, and Vyas Sekar. 2016. Gremlin: Systematic resilience testing of microservices. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 57–66.

[11] Jose-Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2016. An automatic process for weaving functional quality attributes using a software product line approach. *Journal of Systems and Software* 112 (2016), 78–95.

[12] David Jaramillo, Duy V Nguyen, and Robert Smart. 2016. Leveraging microservices architecture by using Docker technology. In *SoutheastCon 2016*. IEEE, 1–5.

[13] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. 2019. Performance modeling for cloud microservice applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. ACM, 25–32.

[14] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, et al. 2018. Service fabric: a distributed platform for building microservices in the cloud. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 33.

[15] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study.* Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

[16] Jörg Kienzle, Anne Koziolek, Axel Busch, and Ralf H Reussner. 2016. Towards Concern-Oriented Design of Component-Based Systems.. In *ModComp@ MoDELS*. 31–36.

[17] Jörg Kienzle, Gunter Mussbacher, Philippe Collet, and Omar Alam. 2016. Delaying decisions in variable concern hierarchies. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 93–103.

[18] Samuel Kounev, Nikolaus Huber, Fabian Brosig, Simon Spinner, and Manuel Bähr. 2017. Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language. *Software Engineering 2017* (2017).

[19] Florian Rademacher, Sabine Sachweh, and Albert Zündorf. 2017. Differences between model-driven development of service-oriented and microservice architecture. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 38–45.

[20] Florian Rademacher, Sabine Sachweh, and Albert Zündorf. 2019. Aspect-Oriented Modeling of Technology Heterogeneity in Microservice Architecture. In *2019 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 21–30.

[21] Ralf Reussner, Steffen Becker, Jens Happe, Anne Koziolek, Heiko Koziolek, Klaus Krogmann, Max Kramer, and Robert Heinrich. 2016. *Modeling and Simulating Software Architectures: The Palladio Approach.* The MIT Press.

[22] Nick Rozanski. 2011. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, 2nd Edition.* Addison-Wesley Professional.

[23] James Rumbaugh, Ivar Jacobson, and Grady Booch. 1999. *The Unified Modeling Language Reference Manual.* Addison Wesley.

[24] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. 2018. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 223–236.