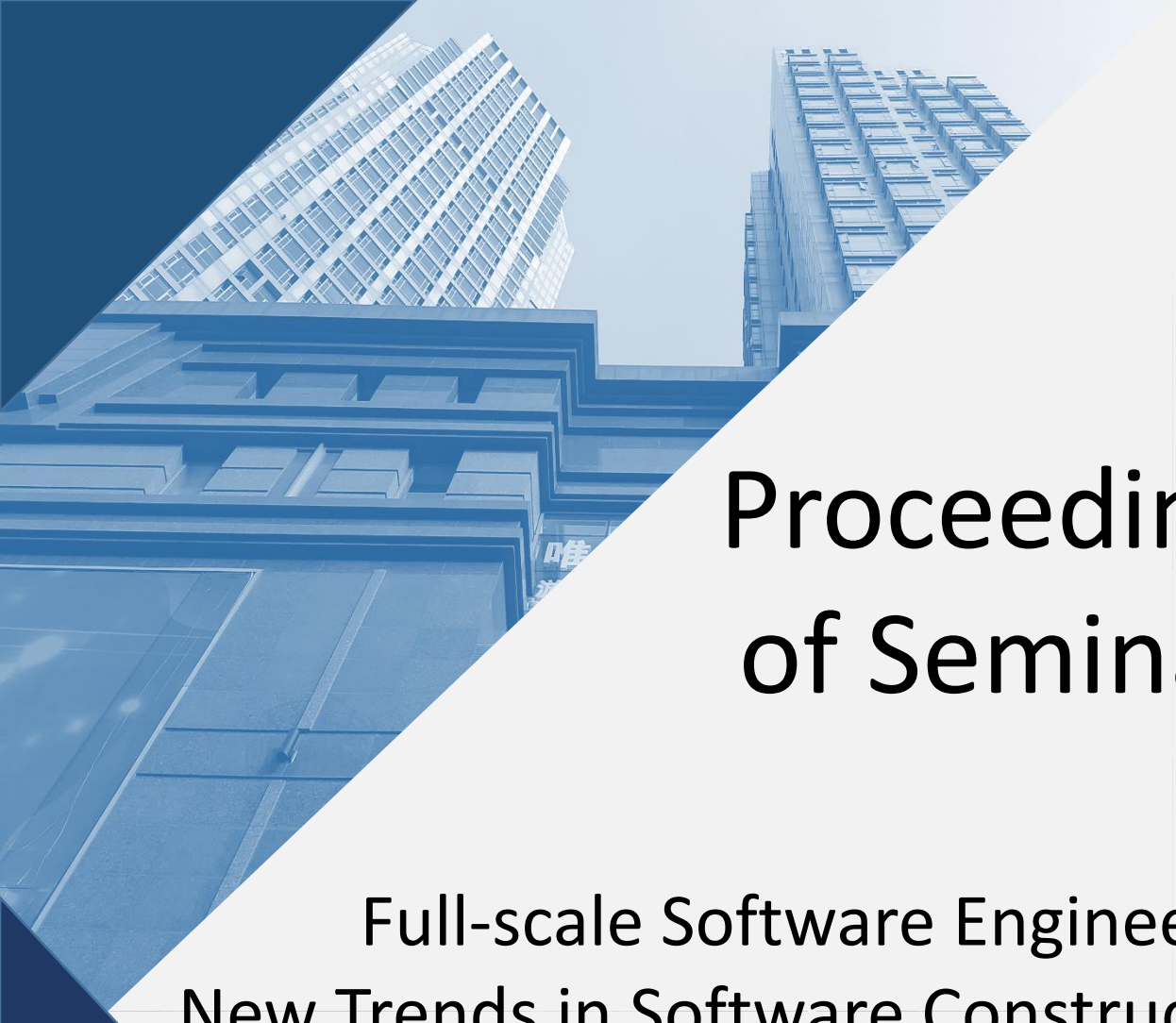

Proceedings of Seminars

Full-scale Software Engineering
New Trends in Software Construction

2022

Editors: Horst Lichter
Peter Alexander
Nils Wild
Selin Aydin
Christian Plewnia
Alex Sabau
Ada Slupczynski

SWC Software Construction | RWTH AACHEN UNIVERSITY

Table of Contents

Cloud Service Selection & Cost Estimation

Sabith Haneef
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
sabith.haneef@rwth-aachen.de

Robert Klingenberg
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
robert.klingenberg@rwth-aachen.de

ABSTRACT

Cloud computing is the delivery of computing services over the Internet. Choosing the best option from all the available cloud services can be challenging. There might not be a single best option. The selection process is further complicated by the fact that the best option can depend on a myriad of factors, such as costs and Quality of Service (QoS) attributes. These factors are not all equally important in every use case and hence the service selection has to be tailored to the needs of the specific case. Estimating the future costs of your choice can also be challenging, as it depends on your chosen service and the predicted usage. In this paper, we conduct a systematic literature review of existing approaches for cloud service selection and future cost estimation. We discuss the various approaches and enumerate relevant features and characteristics in this context. We give an overview of the challenges and drawbacks of these approaches and areas for improvement in the future.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*productivity, programming teams, software configuration management*

Keywords

Cloud Computing, Service Selection, Cost Estimation

1. INTRODUCTION

Cloud computing is the delivery of computing services — including servers, storage, databases, networking, software — over the Internet and has been around for two decades now [20]. It provides developers and businesses the ability to have their solutions up and running with the click of a few buttons or a few API calls. This enables users to easily scale their application. We use the term application here, and elsewhere in this paper, to refer to the software the users wants to deploy to the cloud. These computing services, referred to as cloud services or services in short from here on, are provided by various different cloud providers. Microsoft Azure, Amazon Web Services (AWS), Google Cloud Platform (GCP) are examples of cloud providers to name a few.

The large number of cloud services on offer today is a double-edged sword. While the variety affords flexibility and increases the chances of finding an ideal service, it also adds complexity to the selection process. For example, there are over 50 types of AWS EC2 (virtual compute server) instances[13] alone. Add in the various regions, the various combinations of memory, network bandwidth, storage and CPU configurations and finding the ideal one that meets all the requirements is now a difficult problem. While one could manually search through the various cloud services on offer, it can become tedious and error-prone as the number of services on offer increases. The problem is compounded when the user has to select cloud services for multiple applications. Each application can have different requirements. Cloud services which are suited for one application can be inadequate for another application. Similarly, changing requirements may trigger the need for different cloud service than currently employed. All of these problems call for more automated and systematic approaches to cloud service selection.

Another aspect that has to be considered when choosing and using cloud services is cost. For businesses, costs are an important factor, and therefore it is essential to estimate the costs for applications in advance. For some, it may even be the goal to reduce or minimize the costs associated with the applications.

In this paper, we try to answer the following questions:

- RQ1: What approaches exist for the selection of cloud services?
- RQ2: What approaches exist to estimate the future cost of cloud services?

To this extent, we do a systematic literature review and try to answer these questions from the perspective of the cloud user - in our case, a project manager who wants to select a cloud service for their project. The paper is structured as follows: first, we give an overview of the review process we employed to identify the papers to review. Then, we describe the papers reviewed, the objectives they are optimizing for and the approaches they have taken in solving the problem. We define a set of features and characteristics, both in general and those specific to service selection or cost estimation, and identify which papers provide these features

SWC Seminar 2022 RWTH Aachen University, Germany.

and how. Finally, we discuss the challenges, drawbacks, and shortcomings of the reviewed approaches and provide our conclusions.

2. REVIEW PROCESS

We conducted the systematic literature review based on *Guidelines for performing Systematic Literature Reviews in Software Engineering* [16]. After a preliminary search based upon the research questions relevant keywords emerged:

- cost prediction
- price estimator
- service selection
- future costs

These keywords were supplemented with synonyms and combined to relevant search strings.

2.1 Search terms

The search terms were adapted for the respective digital libraries. The search was conducted on the 05.05.2022 and covered the years 2009-2022. Two digital libraries were used for the systematic literature review: IEEE and ACM. IEEE and ACM are big digital libraries and are recommended for studies in the field of software engineering by Kitchenham [16].

The corresponding search string for the document title: 'Cost prediction' OR 'price prediction' OR future costs' OR 'future price' OR 'future Service selection costs' OR 'future Service selection price' OR 'cost estimator' OR 'price estimator' OR 'Cost estimation' OR 'price estimation'. This is combined with a search string which is searched in the metadata: 'Cloud'.

The search terms are focused to find relevant keywords in the document title and the keyword "cloud" is searched in the metadata (i.e. document title, full text, keywords, affiliation and publisher) to increase the relevance of the results.

2.2 Filters

The search resulted in 73 (IEEE) and 72 (ACM) papers. The number of papers were narrowed down by the following inclusion criteria:

- IC1: papers that conducted cloud service selection
- IC2: cost estimation from the user (of the cloud) perspective

These exclusion criteria were applied:

- EC1: papers with a focus on short time cost predictions, i.e. spot price predictions.
- EC2: papers with a focus on highly specialized services

These filters narrowed the number of papers to 9.

3. OVERVIEW OF APPROACHES

The approaches are grouped into "service selection" and "cost estimation". This grouping is based on the main objective of the approach, 8 approaches had service selection as their main objective and one approach focused only on cost estimation. We provide an overview of the respective approaches and provide context for them. The approaches are compared with a short overview of the commercial tools available for service selection. The chapters following this one are focused on specific features that are important based on the literature. The approaches are discussed in more detail regarding the features they offer. Based on this, the shortcomings and challenges of the approaches are discussed.

3.1 Service Selection Approaches

In this section, we review how the various papers tackle the problem of service selection and discuss the objective of the approach, the inputs, and the outputs.

Cloud service selection is the process of finding a cloud provider who can best satisfy a user's needs [25]. The best cloud provider should meet all the user's needs, referred to as requirements by us, while optimizing one or more objectives - for example minimizing cost of deployment or maximizing network throughput. We define the user as the project manager of the corresponding project. This is due to the impact the cloud service selection may have on the project and the various trade-offs that need to be factored in. We refer to the project manager, whenever we mention user from here on out.

While most of the papers reviewed framed the problem of service selection as that of selecting a service that optimizes a general set of user selected Quality-of-Service (QoS) attributes, a few were focused on specific use-cases and tailored their optimization objective around that use-case.

Soltani et al.[22] proposed QuARAM Service Recommender that adopts a hybrid approach (case-based and MCDM based) for service selection. On receiving application requirements and customer preferences, a case-based recommender is employed to recommend a suitable service using previous deployment information. Use of past deployments can help accelerate the selection process as only a subset of service providers have to be considered which were previously recommended. However, this presents a problem in that if a prior recommendation was bad for whatever reason (maybe the services at that time were limited and now better services are available), the recommendations for similar future services are also affected. The authors do try to mitigate this by having the case-bases updated based on feedback from users. In the absence of any such previous deployments or if the precision of case-based recommendation is below a specified threshold, the problem is modeled as a Multi-Criteria Decision Making (MCDM) problem and solved to find the most suitable service. This approach is an answer to RQ1, but it works best if there are enough cases in the case-base to make it effective.

Abourezq et al. [1] developed the Cloud Services Research and Selection System (CSRSS) to select cloud services that best meet the user's requirements. Initially, user selects the criteria the cloud services must meet. These requirements are then categorized into two - fixed (such as provider's name, OS, etc.) and variable (price, bandwidth, etc.). The fixed requirements are used to query a database and obtain a set of services meeting these requirements. A combination of algorithms were applied to this result set to select the cloud services that offer the best compromise in all the criteria defined by the user. This approach posed a problem in that the final result set was still quite large. For instance, from 50,000 cloud services the final result set contained over 2,500 services, meaning the user still had work

to do to find the ideal service. Rehioui et al.[21] extended CSRSS by adding clustering to operate on the result of the previous step and find a small set of services closest to the user's ideal service. They were able to reduce the size of the resulting set to less than 15 and the user now only had to compare a few services to arrive at their ideal service.

Tan et al.[23] introduced Probabilistic Hierarchical Refinement (ProHR) for optimizing selection of composition of services. They consider an application as a composition of micro-services with defined global QoS constraints and use ProHR to find an optimal composition of services from a pool of competing similar services. ProHR is divided into three stages - preprocessing, probabilistic ranking, and hierarchical refinement. In the preprocessing stage, global QoS constraints are used to prune all services that fail to satisfy the global constraints and help reduce the search space. In the probabilistic ranking stage, the remaining services are ranked according to a combination of two criteria - their QoS values individually and as part of a composition. In the final step, Hierarchical Refinement, an optimal composition of services is selected from the ranked services using Mixed Integer Programming (MIP). The result is a composition of services that maximizes the global QoS value and meeting all global QoS constraints.

Often times, multiple applications need to be deployed at multiple locations in order to provide the end-users (here we mean users using the application) the best possible experience in terms of minimum response times. However, locations closer to users may also cost more money due to a multitude of factors (like data centers in metropolitan cities, where most users might reside, might be costlier than in smaller towns). Tupsamudre et al. [24] tackled this problem (also known as Web Services Location Allocation Problem or WSLAP) subject to minimizing total deployment cost and network latency. To this end, they proposed a Divide and Conquer approach whereby WSLAP is decomposed into sub-problems of finding location allocation for each application and then the solutions are merged to arrive at the final allocation for all applications. They employ evolutionary algorithms to solve the optimization sub-problems and their own novel combine algorithm for the merge step. The result is a configuration which tells the cloud user where to deploy which application, such that the total deployment cost is minimal while also minimizing the latency for the majority of end-users.

Another popular use-case of cloud services is using it for data storage. And for many, loss of data is unacceptable and replication is a necessity. However, like with all cloud services, money is a big factor and finding storage that meets requirements and is cost effective is a challenge. Chang et al. [4] focused on this selection of service providers for replicated cloud storage. They proposed two probability based algorithms for service selection that maximized availability with a given budget. They characterized service providers with two properties - price and failure probability - and used these to find a selection of services that fits within the user's budget and that minimized the probability of data loss. Their approach helps answer RQ1 for a particular type of cloud service, namely cloud storage. A drawback is that it requires knowledge on failure probability of the services, which is not always the case as some providers does not provide this information.

Serverless computing is a cloud computing execution model where users can execute application logic as functions and all operational concerns are delegated to the cloud platform. To implement complex business functionality, individual functions are composed into serverless workflows. Estimating the cost of these workflows is rather complex, as the response time and hence the cost of a function depends on its input parameters, which are propagated from prior functions within the workflow. Eismann et al. [8] proposed a methodology to predict the cost of such workflows, taking into account the influence of input parameters. They modeled the distribution of a function's response time and output parameters and used it to estimate the cost of the workflows. The cost estimation then enables users to evaluate and compare workflow alternatives and optimize existing workflows.

Chard et al. [5] developed a Scalable Cost-Aware Cloud Infrastructure Management and Provisioning (SCRIMP) tool. This modular tool focuses on reducing the needed knowledge to select services, improve the performance based on profiling and predictions. This means that user defined policies (i.e. preferences of the user) are used together with a profiling module and a prediction module to provision infrastructure. These profiles are a performance measure for specific applications on specific cloud instances. The profiles are created by an "offline profiling service" (service does not mean "cloud service" in this context, but is the term used in the approach) and augmented by monitored data during deployment. Based on the profiles and price prediction, a list of possible cloud instances is filtered. SCRIMP then bids based upon the price prediction together with the user profiles. The deployment is monitored and changes to the provisioning are made continuously. To summarize the inputs and outputs of this approach: The application/job and user preferences(e.g., preferences for spot-market instances (see Dynamic market prediction6.3), or specific instance types[5]) are needed. The application is then deployed to the chosen service.

Wang et al. [26] focus on mobile micro clouds and their future costs. This approach can be adapted for a more generic service selection. The focus here lies on changing spatial requirements, i.e. applications need to be migrated to accommodate the changed location of the end-user. The input for this approach is composed of discovered cloud services and estimated costs for migration of a application. The result depends on the used algorithm, the paper provides an offline and online algorithm. In the case of the online algorithm the result is the mapping from application to service for the next time step. For the offline algorithm, the result is the mapping for a certain time horizon, which can be changed accordingly. In both cases this mapping corresponds to the smallest expected costs.

3.2 Cost Estimation Approach

We found a paper that only provides a solution for RQ2 and not for a service selection. We present this separately.

Aoshima et al. [2] focuses on estimating the costs for deploying applications early in the design phase. Speed of this estimation is preferred to accuracy. This approach does not provide service selection. For this rough cost estimation several details are needed. The rough computational requirements are needed, i.e. how many resources are needed to handle an event, how many events occur and what other applications are needed. Resources denote the needed software and hardware, this depends on which cloud service is used.

Resources can include cloud instances, storage, databases, analytics and other services which are provided by cloud service providers. The costs are modelled as a directed acyclic graph (DAG) where each node corresponds to an application and an edge represents a relationship between applications, while the direction shows the direction of a request. The graph is used to define the relations between the applications, but for the cost estimations formulas are used (and not graph algorithms). Costs which are associated with networks i.e. requests between applications are considered in this approach. The computational requirements to handle an event are also needed. The user has to select a cloud provider and use the price list as an input for this approach. This is not necessarily the most cost efficient provider, but this is not further considered in this rough pre-design cost estimation. Additionally non-linear costs can be modelled. This can be useful when the resource needs depend on the workload, i.e. for a high workload should another instance be used compared to a low workload.

3.3 Commercial Tools

The landscape of commercial search and selection tools for cloud services is rather uninspiring. In 2016, Eisa et al.[7] did an analysis on the available tools at the time which were Intel Cloud Finder, Cloudorado[6], and RankCloudz[18]. Of these, Intel Cloud Finder is not available anymore. RankCloudz has not been updated since at least 2017, the website is not up to modern web standards (like lack of responsive and mobile friendly UI for example) and only feature five cloud providers. Cloudorado is online and working and has improved since the analysis was done. It divides cloud services into three categories- Cloud Server, Cloud Hosting and Cloud Storage. It also has a page tabulating the various cloud providers and their features. Users can specify their requirements and a list of providers matching the requirements is returned, sorted from cheapest to most expensive. Cloudorado limits its search to under 15 providers and the price provided was less accurate than in the respective provider's pricing calculator. For example, on June 1 2022, the price mentioned in Cloudorado for an EC2 instance (c5x.large instance type, US East region, 4 vCPUs, 8 GB RAM, 17 ECU or EC2 Compute Units, 20 GB SSD Amazon Elastic Block Storage) was $75.00 per month for 1 year while it was $ 80.11 in AWS pricing calculator[14]. Other commercial tools (CloudHarmony[15], cloudcomparisontool[17], CompareCloud[10] to name a few) offered limited capability and customizability and were also restricted to the major cloud providers in their comparisons.

In summary, while the cloud services on offer has grown tremendously in the past decade, commercial tools currently available for cloud service comparison and selection have failed to keep pace with it. While existing tools do help users in some capacity, they are all severely lacking in capability (like ability to mention fine-grained requirements and constraints), customizability (like add or remove features or providers they prefer) and usability (like modern mobile friendly and responsive UI).

4. GENERAL FEATURES

Several important features emerged while comparing the different sources. Some features are distinctive features that apply in service selection, and several are useful in the context of cost estimation. A few features could also be classified to be relevant for both topics. We grouped them together in this section. Table 1 gives a quick glance at the general features.

4.1 Type of Cloud Service

Cloud services are generally classified into three — Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Recently, with the rise of serverless computing, new paradigms such as Function-as-a-Service (Faas) and Backend-as-a-service (BaaS) have also emerged[20].

While most of the papers reviewed does not make any such distinction in their selection process, few focus specifically on one of these types. The advantage of focusing on one is that it allows to make use of characteristics specific to that class and may help enhance the effectiveness of the approach. For example, Eismann et al. [8] focuses on FaaS and specifically takes use of the fact cost models employed by most FaaS providers is directly related to how long each invocation of the function takes. We also observe that none of the approaches focus specifically on PaaS, which could be an area to focus for future research.

4.2 Monitoring

Monitoring can mean different things in the context of service selection and cost estimation, we define monitoring as the application monitoring after deployment to assess the workload in the virtual machines (VMs). This can be used as an input for online optimization 4.6 which need to know current workloads to work. Monitoring in the context of application profiling is further discussed in the section *Profiling* 6.2.

Chard et al. [5] uses monitoring in several steps. It is mainly used in monitored queues for different cloud providers. Jobs are submitted into a queue to be deployed. After the application is deployed, the execution is monitored until the application is finished. This is used in this approach for VM scaling 4.4 and online optimization 4.6.

4.3 Multicloud

The services of different cloud providers differ in their price and quality of service (QoS). Often times, the optimal selection involves making use of multiple cloud service providers. It is more complicated for an service selection/cost estimation approach to support multiple cloud providers. This is due to the increased search space, the problem of comparing different services and differences in pricing structure.

The technique by Tan et al.[23] allows this composition of multi-cloud providers for IaaS and SaaS selection, while Eismann et al. [8] and Chang et al. [4] enable this for serverless cloud functions and cloud storage providers respectively. Chard et al.[5] support AWS, Globus galaxies and Cloud Kotta the support for other commercial/academic cloud platforms will be added. Aoshima et al. [2] calculates rough cost estimates and does only need the costs of different cloud providers to work in a multi cloud context.

4.4 VM scaling

As an application usage can change overtime, its needs and requirements can change as well. Due to these changes resources that were optimal at the beginning may not be sufficient and additional resources may be required. The scal-

Table 1: General Features

Paper	Type of Cloud Service	Monitoring	Multicloud	VM Scaling	Network Optimization	Online Optimization
Soltani et al.[22]	General	Yes	Yes	No	Yes	No
Tan et al.[23]	IaaS/SaaS	No	Yes	No	Yes	No
Rehioui et al.[21]	General	No	No	No	No	No
Tupsamudre et al. [24]	General	No	Yes	No	Yes	No
Chang et al. [4]	IaaS	No	Yes	No	No	No
Eismann et al. [8]	FaaS	No	Yes	No	No	No
Chard et al. [5]	IaaS	Yes	Yes	Yes	No	Yes
Wang et al. [26]	General	No	No	No	Yes	No
Aoshima et al. [2]	General	No	Yes	No	Yes	No

ing of resources to accommodate the new, changing requirements can be divided into horizontal and vertical scaling. Horizontal scaling means changing the number of virtual machines and vertical scaling means changing of the virtual machines size. This change can mean that the number of VMs/VM size is increased or decreased.

Chard et al. [5] utilizes Profiling 6.2 and Monitoring 4.2 to continuously update application profiles. These profiles determine the needed resources for this application. This has the advantage of heterogeneous horizontal scaling, i.e. the instances for an application are not necessarily of the same instance type.

4.5 Network optimization

Network optimization is a closely linked to Quality of Service (QoS) optimization. It is mainly done to reduce the latency for end-users of the applications and reduce the request time between applications. End-users refer to the people which use the applications which are deployed in the cloud. Reduced latency can lead to increased satisfaction for users. This leads to a need to model the communication between applications. Aoshima et al. [2] model the requests an application sends to another application, additionally the needed bandwidth and CPU resources to handle the requests are modelled. This is used as a rough estimate for the resulting costs of the network communication of the deployed applications.

4.6 Online optimization

The approaches can be categorized broadly into two different groups, based on the point in time when the selection/estimation is conducted. Some approaches conduct the selection/estimation once and select services based on the results (offline algorithm). Other approaches conduct a continuous selection/estimation, we call this online optimization or online algorithms. They usually are computationally more expensive regarding the selection/estimation but can react to changing demand, market conditions or other relevant changes.

Chard et al. [5] incorporates a queue based service selection. This approach can work with applications that have limited execution time and applications that are running continuously. Request to deploy Applications can arrive at anytime. This necessitates an online optimization to select services for new arriving applications. Additionally service selection and cost estimation methods are used for VM scaling.

5. SERVICE SELECTION FEATURES

We enumerate several features which we believe service selection approaches should strive to provide. We discuss what we mean by these features, why they need to be a part of any approach and the approaches presented in literature that provide these features. A quick overview is provided in Table 2, excluding approaches that do not deal with service selection.

5.1 Service Selection

The result of service selection can be put into two categories — a single candidate chosen by the selection process or a list of viable candidates from which the user can select the one they see fit. In either case, the selected service or service(s) meets all the requirements set forth by the user and it is up to the user to determine which approach they want to use.

Solutions proposed by Soltani et al.[22], Tan et al.[23], Tupsamudre et al. [24] and Chang et al. [4] fall into the first category. Solution of Abourezq et al. [1] and its extension by Rehioui et al.[21] offer the user a set of service providers that is as close to the ideal service needed by the user but leave the final decision to the user itself.

5.2 Deployment

We mentioned already how manually selecting services can become tedious, especially as the number of services increases. Deploying to the selected service manually again poses this same problem, and having the selection service automatically do the deployment will go a long way in solving this problem.

Of all the papers reviewed, only the QuARAM Framework by Soltani et al.[22] has the capability to take the result of the service selection process and deploy to the selected service. In QuARAM, the recommended provider configuration is passed onto a deployment engine which takes care of the deployment. All other papers only return the result of the selection process back to the user and the obligation is on the user to then deploy their applications.

5.3 Incorporates User Feedback

User feedback on the results of service selection, such as whether the selected service matched the expectations of the user or whether the quality of service was not as advertised by the provider etc., can provide valuable insights. Incorporating such feedback into the service selection can improve the results for future queries.

Soltani et al.[22] were able to incorporate user feedback to their case based recommender and in doing so were able to improve the quality of future recommendations as measured by their experiments.

Table 2: Service Selection Features

Paper	Service Selection	Deployment	Incorporates User Feedback	Service Composition	Performance Optimization
Soltani et al.[22]	Yes	Yes	Yes	Yes	Yes
Tan et al.[23]	Yes	No	No	Yes	Yes
Rehioui et al.[21]	Yes	No	No	No	No
Tupsamudre et al. [24]	Yes	No	No	Yes	Yes
Chang et al. [4]	Yes	No	No	No	No
Eismann et al. [8]	Yes	No	No	Yes	No
Chard et al. [5]	Yes	Yes	No	No	No
Wang et al. [26]	Yes	No	No	No	No
Aoshima et al. [2]	No	No	No	No	No

5.4 Service Composition

We define service composition to mean an application (also referred to as a composite service) composed of multiple services (called component services) which communicate with each other to facilitate functionality. Many companies now employ this type of architecture instead of one big monolithic application (for example at Netflix [9], Uber[12], Spotify[11] etc.). Composite service selection aims to find suitable cloud services for such composition of component services. For example, the user may not care about the latency in communication between the component services so long as the total end-to-end latency of the composite service remains under the specified limit. This may present an opportunity to select cloud services which may not have the best latency but has other factors (for example cost) that make it a better choice. The selection process should then take these factors into consideration.

Tan et al.[23] solves this problem by defining global QoS constraints for the service composition, using their methodology to rank services for each component service that satisfy the constraints and then finding an optimal composition from these ranked services. Soltani et al.[22] used their case-based recommender to identify an optimal composition from prior deployments. If the case-based recommendation fails (as can happen if there are no similar cases in the case-base), suitable cloud services are found for each component service separately and a consolidator is used to find the optimal composition.

5.5 Performance Optimization

Regardless of the different optimization objectives and use-cases, service selection is a search problem. The search space gets bigger each year with the number of available cloud services growing. A bigger search space increases the computational effort. Combine that with the various choices for regions, CPU, network bandwidth, storage and memory configurations, which leads to an even bigger search space. For example, as of June 2022 in Microsoft Azure[19], there are over 50 VM instance types and over 40 regions. The problem is more pronounced in the case of service composition. A composite service which is made up of 10 component services and each of these is deployed on one of the 2000 options (50 types $\times$ 40 regions). This results in 2000^{10} possible options and exploring all these is often impractical. In this case the solution space is already big, but if we consider other cloud service providers, scaling of VM instances or other features that are considered in this paper, the solution space gets even bigger. Therefore, different techniques and optimizations need to be employed, for example to reduce the search space, and ensure that service selection is performant.

Use of the case-based recommender in Soltani et al.[22]'s approach help speed up the selection process since all available services are not searched, but rather a subset of services which were previously recommended for similar applications. Tan et al.[23] cut down on the search space in their preprocessing stage by eliminating all services that cannot satisfy the global constraint as well as removing all dominated services (for a service to be dominated means there exists another service which is better or equal in all aspects than this service, this also called "Skyline operator"). Similarly Abourezq et al. [1] use the Skyline[3] method to cut-down on the possible solutions before any optimization algorithms are applied.

6. COST ESTIMATION FEATURES

Cost estimation approaches can be characterized by several features. These are mainly concerned with modelling capabilities and can depend on other more general features. We use the term cost as the monetary cost a user of cloud services has to pay. This includes the costs related to data processing i.e. the instance cost and the data transmission costs i.e. API calls to applications. The costs for the data processing includes the setup of the application and the actual workload of the application. The cost for data processing can be higher than necessary, if an instance is oversized compared to the actual need of the application. The papers and the cost estimation features that they offer is summarized in Table 3.

6.1 Cost estimation

Cost estimation is needed to know the costs of certain deployments or to have estimations which are used for the service selection.

Eismann et al. [8] worked on a methodology to estimate the cost of serverless workflows. One of the factors in deciding the cost of a serverless function is its execution duration which is directly dependent on the function's input. For example, a function which compresses an image will have longer execution time for a larger image. And in a workflow, the input of one function is the output of another. The approach in this paper involved predicting the distribution of a function's response time and output parameters based on it's input parameters and then using that to model the cost of various workflows.

Chard et al. [5] use an online optimization approach. The cost estimation works by approximating resource requirements of the application (see Application Profiling 6.2). All candidate instances are combined with a cost function, this depends on the dynamic market prediction (See Dynamic market prediction 6.3), the advertised/negotiated prices and

Table 3: Cost Estimation Features

Paper	Cost estimation	Application Profiling	Dynamic Market Prediction	Non-linear Cost Modeling
Soltani et al.[22]	No	No	No	No
Tan et al.[23]	No	No	No	No
Rehioui et al.[21]	No	No	No	No
Tupsamudre et al. [24]	No	No	No	No
Chang et al. [4]	No	No	No	No
Eismann et al. [8]	Yes	No	No	No
Chard et al. [5]	Yes	Yes	Yes	No
Wang et al. [26]	Yes	No	No	No
Aoshima et al. [2]	Yes	No	No	Yes

the preferences of the user. These preferences can include: preference for spot market instances, a specific instance type, or other preferences. Suitable instances are selected with their associated costs, with the instance cost and the expected duration the costs can be calculated. Wang et al. [26] focus on cost estimation with applications that often need to migrate (to another location, not necessary another cloud provider). This migration happens to accommodate spatial requirements of the end-user, i.e. the people which use the application. The costs are divided into local costs and into migration costs. The local costs include the costs for data transmission and data processing, they depend on the end-user location, network condition and instance cost. The migration cost denotes the cost associated with the migration of an application to a service at another location. For both costs the information are gathered directly from the cloud provider or are based on historical data.

6.2 Application profiling

For service selection and cost estimation, it is essential to have an estimation for the resources which are needed. These estimations are especially important for the cost estimations, as the resource type and quantity determines a substantial part of the cost.

There are several methods to determine the resource needs for an application. User estimations are often incorporated, but are often inaccurate [5]. Cloud service providers describe resources in different formats, which makes it harder to compare them [5]. The performance of an application can depend on the specific resources, which makes it even harder to know the resource demands. Another method which was found in the literature is application profiling. This is based on running the application and measuring performance and resource utilization. These resource requirements are summarized in a profile. We define a profile the same as Chard et al. [5]: "[...] a concise description of the performance and CPU, memory, network, and disk requirements of an application under different environments and scenarios."

Only the approach from Chard et al. [5] incorporates such a profiling. This approach focuses on the provisioning of applications, therefore only virtual machine (VM) instances are considered in the profiling. The application profiling is conducted in two steps: 1. An offline step where the application is deployed on several instances to estimate the resource requirements.2. The application is continuously monitored to improve the estimation (profile).

6.3 Dynamic market prediction

Market prices are depended on various factors. There are the advertised prices per instance and unit of time for the different cloud providers. These can be divided into "pay as you go" and "reserved" instances. Reserved instances are cheaper, but they are only available with long term contracts (usually one year or more) These prices and availability are guaranteed. Cloud providers seek to increase the utilization of their cloud and incentives the usage of unused instances with lower prices, so-called spot prices or dynamic market prices. These instances are bid upon by various users and have lower prices, but the actual usage can not be guaranteed. This happens if the bid is not won. Another disadvantage is that the instance can be withdrawn due to changes in the market, i.e. the usage of the base price instances increases. Service selection and cost estimation approaches can incorporate the capability to handle/use these dynamic prices to lower the costs for the user.

Chard et al. [5] use an approach which sets a minimum bid price which provides a probabilistic guarantee that the bid is won. The is done by predicting the spot market price of the instance and more that this prediction. This system is based on a time series of upper bounds of market prices. This is continuously updated for increased accuracy.

6.4 Non Linear Costs

Costs are often modelled on the assumption of linear scaling, i.e. for an increase in the workload of 50 % roughly 50 % more resources of the same type are needed. The same resource type should cost the same when more is needed. This is not the case when non linear workloads are introduced. These non linear workloads require different resources depending on the workload. An application could be optimized in such a way that for low usage a CPU is utilized and with increased utilization the workload is shifted towards a GPU. This leads to a change in type of instances that is needed depending on the workload of the application.

Aoshima et al. [2] provide this capability with a threshold based non-linear cost modeling. For this, changes in their cost matrix are made.

7. DISCUSSION

The goal of this paper was to give a systematic literature review to present the state of the art regarding the research questions:

- RQ1: What approaches exist for the selection of cloud services?
- RQ2: What approaches exist to estimate the future cost of cloud services?

Based on this literature review, 9 relevant papers were identified which use very diverse approaches to tackle ser-

vice selection/cost estimation. Below we present issues we identified in the approaches we presented in this paper and then we discuss the limitations of our paper.

7.1 Challenges, Drawbacks, and Shortcomings of current approaches

A big drawback of all the existing approaches is the lack of clarity on where the data for analysis comes from. All the approaches discussed make extensive use of data about the QoS attributes of the various cloud services like the latency, throughput, guaranteed up-time etc. They assume the availability of said data for the selection process, however this may not be the case all the time for all the services. Similarly, there is no single source where one can go and get a list of all available services. This is an issue in service discovery and selection - a lack of a global service registry that is up-to-date on information like all the available cloud providers, their features and characteristics. Owing to these challenges in getting data, some papers (like Rehioui et al.[21]) resorted to using dummy data for their analysis.

Another aspect almost all the papers fail to incorporate are the non-quantifiable requirements. For example, certifications and compliance such as track record, trust worthiness, privacy of customer data and customer service are non-measurable features of the service provider that may have a big impact in whether a user wants to choose a particular service. However, current approaches predominantly focus on quantifiable aspects when selecting services.

Most of the papers on service selection employ offline optimizations. These can not react to changing requirements. With an offline optimization the service selected at the beginning may be inadequate to satisfy the updated requirements. More approaches should implement online optimizations to tackle these problems. Additionally, as new cloud services are introduced, they may be better suited that previous services. An approach should discover these and consider it for the future service selection.

The approaches that are considered in this paper also did not utilize "reserved instances", i.e. cheaper instances but with a long term commitment. This is a field where cost reductions could be made. This needs to be explored further.

It can be seen that no approach has an implementation for all identified features. Most papers focus just on service selection or cost estimation and do not implement the other. Some features are rarely observed, these include monitoring, VM scaling, online optimization, deployment, incorporating user feedback for future improvements, service composition, Cost estimation, Application Profiling, Dynamic market prediction and Non linear cost modelling. This observation does not mean that an approach has to implement all features to be a viable approach. Some features have only niche usages and are less important than other features. Examples for this are "Incorporates User Feedback" and "Non-linear Cost Modeling", especially the non-linear cost modeling is only useful if the user wants different instance types for their respective workload.

Future research should therefore focus on utilizing VM scaling, online optimization, service composition in a multi cloud environment. Another goal should be to incorporate dynamic market pricing into the service selection, combined with the capability of non linear cost modeling.

7.2 Limitations

These findings are based upon a limited number of keywords and digital libraries. This study is not a comprehensive overview of all relevant papers on this topic. Other relevant paper can emerge with other keywords and different sources. The exclusion criteria EC1, EC2 (See Filter 2.2) could have filtered out papers which focused on the respective topic, while implementing a more general approach to service selection or cost estimation. While we tried to appeal to a wide audience, some content may still be at a deeper level where some prior knowledge might be necessary to understand the text. Conversely, some parts of the paper might be too high-level and may have needed deeper explanations.

8. SUMMARY

In this paper, we provided a systematic literature review based upon the research questions RQ1 and RQ2. We mentioned several reasons why a systematic and automated approach to service selection is needed. We found several approaches for service selection and cost estimation. We identified several features any service selection or cost estimation approach should have, explained why these features were important and gave an overview of how the reviewed approaches provided some of these features. We found that no approach implemented all the identified features. This should be possible, but is likely impractical. We also documented the various short comings of these approaches. The number of approaches that provide an easy to use and powerful real time solution is lacking. Many features have a trade-off between accuracy, capabilities and computational complexity/speed. An approach should be selected depending on the preference of the user.

9. REFERENCES

[1] M. Abourezq and A. Idrissi. A cloud services research and selection system. In *2014 International Conference on Multimedia Computing and Systems (ICMCS)*, pages 1195–1199, 2014.

[2] T. Aoshima and K. Yoshida. Pre-Design Stage Cost Estimation for Cloud Services. *Proceedings - 2020 IEEE 44th Annual Computers, Software, and Applications Conference, COMPSAC 2020*, pages 61–66, 2020.

[3] S. Borzsony, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings 17th International Conference on Data Engineering*, pages 421–430, 2001.

[4] C.-W. Chang, P. Liu, and J.-J. Wu. Probability-based cloud storage providers selection algorithms with maximum availability. In *2012 41st International Conference on Parallel Processing*, pages 199–208, 2012.

[5] R. Chard, K. Chard, R. Wolski, R. Madduri, B. Ng, K. Bubendorfer, and I. Foster. Cost-aware cloud profiling, prediction, and provisioning as a service. *IEEE Cloud Computing*, 4:48–59, 2017.

[6] Cloudorado. Cloud Computing Comparison Engine. `https://www.cloudorado.com/cloud_server_comparison.jsp`. Retrieved June 1, 2022.

[7] M. Eisa, M. Younas, K. Basu, and H. Zhu. Trends and directions in cloud service selection. In *2016 IEEE*

Symposium on Service-Oriented System Engineering (SOSE), pages 423–432, 2016.

[8] S. Eismann, J. Grohmann, E. van Eyk, N. Herbst, and S. Kounev. Predicting the costs of serverless workflows. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, page 265–276, New York, NY, USA, 2020. Association for Computing Machinery.

[9] J. Evans. Mastering Chaos: A Netflix Guide to Microservices. `https://www.infoq.com/presentations/netflix-chaos-microservices/`, 2016. Retrieved June 1, 2022.

[10] I. F. Public Cloud Services Comparison. `https://comparecloud.in/`, 2022. Retrieved June 1, 2022.

[11] K. Goldsmith. Microservices at Spotify. `https://gotocon.com/berlin-2015/presentation/Microservices%20@%20Spotify`, 2015. Retrieved June 1, 2022.

[12] E. Haddad. https://eng.uber.com/service-oriented-architecture/. `https://eng.uber.com/service-oriented-architecture/`, 2015. Retrieved June 1, 2022.

[13] A. W. S. Inc. Amazon EC2 Instance Types - Amazon Web Services. `https://aws.amazon.com/ec2/instance-types/`, 2022. Retrieved June 1, 2022.

[14] A. W. S. Inc. Amazon Pricing Calculator. `https://calculator.aws/#/createCalculator/EC2`, 2022. Retrieved June 1, 2022.

[15] C. Inc. CloudSquare Provider Directory. `https://cloudharmony.com/cloudsquare/`. Retrieved June 1, 2022.

[16] B. A. Kitchenham. Systematic review in software engineering: Where we are and where we should be going. In *Proceedings of the 2nd International Workshop on Evidential Assessment of Software Technologies*, EAST '12, page 1–2, New York, NY, USA, 2012. Association for Computing Machinery.

[17] Z. Live. Cloud Comparison Tool. `https://cloudcomparisontool.com/`. Retrieved June 1, 2022.

[18] R. T. LLP. RankCloudz Online - Cloud Comparison and Evaluation. `https://www.rightcloudz.com/RankCloudzOnline/`, 2017. Retrieved June 1, 2022.

[19] Microsoft. Azure Products by Region. `https://azure.microsoft.com/en-us/global-infrastructure/services/?products=virtual-machines&regions=all`, 2022. Retrieved June 1, 2022.

[20] Microsoft. What is Cloud Computing? `https://azure.microsoft.com/en-us/overview/what-is-cloud-computing`, 2022. Retrieved June 1, 2022.

[21] H. Rehioui, A. Idrissi, and M. Abourezq. The research and selection of ideal cloud services using clustering techniques: Track: Big data, data mining, cloud computing and remote sensing. In *Proceedings of the International Conference on Big Data and Advanced Wireless Technologies*, BDAW '16, New York, NY, USA, 2016. Association for Computing Machinery.

[22] S. Soltani, K. Elgazzar, and P. Martin. Quaram service recommender: A platform for iaas service selection. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, UCC '16, page 422–425, New York, NY, USA, 2016. Association for Computing Machinery.

[23] T. H. Tan, M. Chen, J. Sun, Y. Liu, E. André, Y. Xue, and J. S. Dong. Optimizing selection of competing services with probabilistic hierarchical refinement. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 85–95, New York, NY, USA, 2016. Association for Computing Machinery.

[24] H. Tupsamudre, S. Saurabh, A. Ramamurthy, M. Gharote, and S. Lodha. A divide and conquer approach for web services location allocation problem. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '21, page 1346–1354, New York, NY, USA, 2021. Association for Computing Machinery.

[25] M. Vakili, N. Jahangiri, and M. Sharifi. Cloud service selection using cloud service brokers: approaches and challenges. *Frontiers of Computer Science*, 13:599–617, 2019.

[26] S. Wang, R. Urgaonkar, T. He, K. Chan, M. Zafer, and K. K. Leung. Dynamic service placement for mobile micro-clouds with predicted future costs. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):1002–1016, 2017.

Identifying Deployment Patterns in MLOps Platforms

Keven Hu
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
keven.hu@rwth-aachen.de

Lennart Holzenkamp
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
lennart.holzenkamp@rwth-aachen.de

ABSTRACT

Machine Learning (ML) continuously becomes more used and thus an efficient and standardized set of development and deployment practices and concepts in the form of **MLOps** is sought after. MLOps is not yet as standardized as general DevOps. Hence, many organizations struggle to deploy their machine learning models to production.

Therefore, we analyze the **deployment** process of machine learning models which are specified in the documentation of three cloud providers that offer machine learning tools. Their differences and similarities in the deployment process are highlighted and compared. The possible options to use a deployed model are also explored.

Among all three different machine learning platforms that are analyzed regarding their deployment process, three common components needed for deployment are found. All platforms deploy to REST-API endpoints. Only one platform requires additional components for the deployment of machine learning models. Therefore, a general deployment pattern of all three tools can be specified.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*deployment, MLOps, Machine Learning, Machine Learning Platforms*

1. INTRODUCTION

Machine Learning is becoming increasingly widespread in a broad variety of fields. From personalized media recommendations to conversational AI agents, machine learning seems to become even more important in the future for researchers as well as companies[39] [33]. Moreover, with this increasing set of potential applications for machine learning, the financial and ethical cost of potential accidents rises too. Whether it is about setting the ticket price on a travelling platform too low or too high or showing only engaging media like conspiracy theories or hate speech to social media users, these accidents incur an unwanted cost[41] [14]. Therefore, researchers and companies are interested in reducing the risks of such accidents by ensuring a safe workflow during the development and deployment of machine learning models. Therefore MLOps practices support developers. At the same time, the management and life cycle of machine learning models should be shorter and more efficient for higher profitability [41]. Additionally, many companies struggle to bring their machine learning models to production which can be improved a lot with MLOps [16][17], especially, by following deployment standards which may be retrieved from patterns among the current leading platforms.

To reach a shorter, safer and more efficient life cycle and management of machine learning models the concept of MLOps has been invented. Machine Learning Operations (MLOps) describes a set of practices to develop, analyze, maintain, deploy and publish machine learning solutions including data-generating, data cleansing and data labelling. The term MLOps is derived from DevOps (Development & Operations) but is not yet as standardized as DevOps is.

MLOps uses well-known concepts of DevOps like Continuous Integration and Continuous Delivery while also introducing new ones like Continuous Training in order to meet the specific requirements of machine learning development and management[41].

Many cloud providers offer MLOps platforms for developing machine learning solutions like Azure Machine Learning or Amazon SageMaker. They offer the ability to gather data, monitor deployed models or train models on a platform. Therefore, a lot of processes along the machine learning life cycle have specifications within these platforms according to the platform holders. This includes the deployment process of machine learning models, which components are necessary for the deployment and how inference requests can be sent[31] [1].

Therefore, four research questions arise:

- R1: What is the process for machine learning model deployment on single MLOps-platforms?
- R2: How do the processes differ between platforms and what do they have in common?
- R3: What model format is needed for deployment?
- R4: What knowledge do developers need in order to deploy a model on the platform?

SWC Seminar 2022 RWTH Aachen University, Germany.

2. BACKGROUND

Machine Learning.

A description would be to call machine learning the designing of accurate and efficient inference and prediction algorithms which use training data samples and are based on concepts of computer science, statistics, probability and optimization. These algorithms are applied to various learning problems such as classification, which assigns categories to objects, ranking, which orders objects according to some criterion, or clustering, which partitions objects into homogeneous regions, for example. The aforementioned learning problems abstract real-world problems like classification, which can be used for speech recognition, ranking, for ordering web search results, or clustering, for identifying communities in social networks.

An artefact called a model, which is trained on training samples, can evaluate new data in processes called prediction and inferencing. The model can make inferences online, sometimes also called real-time, or in batches. An online model receives data points in real-time which have to be evaluated one at a time while a batch model receives large data sets of data points for evaluation at once[32].

DevOps.

DevOps can be used in highly iterative software development processes such as Scrum or Extreme Programming and offers the realization of three concepts: Continuous Integration, Continuous Delivery and Continuous Deployment. Continuous Integration ensures that after each iteration a ready-to-run application is available which might not have been tested yet. Furthermore, Continuous Delivery offers a candidate for testing and then release after each iteration. Finally, Continuous Deployment ensures that due to Continuous Integration and Continuous Delivery a tested application is deployed. These concepts help realize automation, for example, in testing[18].

MLOps.

MLOps is about standardizing and shortening the life cycle and its management of machine learning models. The life cycle of a machine learning model consists of the following steps:

- Defining the goal and purpose of the machine learning model,
- gathering data and developing and training the model,
- preparing the model for production,
- deploying the model to production and monitoring the model for feedback used in future iterations.

Automation of several life cycle steps plays a big part in shortening and standardizing the life cycle. Thus, Continuous Integration and Continuous Delivery from DevOps are used, which are necessary for the software-related challenge of automating, for example, the testing of machine learning models[41]. Furthermore, Continuous Training is a new concept from MLOps due to the unique training requirements of machine learning models. It ensures that a model can always be automatically trained on newly obtained data by requiring some sort of data validation and model validation[25].

blue/green deployment.

During the deployment of services, that do not tolerate downtime, it is crucial to have a technique that enables rapid switching between software versions. Therefore one might deploy the new software version to a different location (e.g., different IP address, different port number). Once the new deployment is ready, the routing can be redefined so that the old access point (e.g., IP address, port number) is used for the new deployment. Some tools offer gradual traffic shifting between deployments.

3. METHODOLOGY

The deployment processes of three ML platforms, Microsoft Azure Machine Learning, Amazon SageMaker and Google Cloud Vertex AI, are compared. In the end, the differences and similarities of the deployment processes are analyzed. If there is a common pattern, it is synthesized and described.

The following parts of the deployment processes and use of the deployed services are analyzed and later compared: Expected *structure* of the given machine learning model which should be deployed, platform defined *steps and components* of the deployment process and the techniques how to **access** a deployed model as a client.

For this research, the documentation of the selected platforms is considered and the deployment part, as described before, is examined for further analysis and comparisons.

Microsoft Azure Machine Learning, Amazon SageMaker and Google Cloud Vertex AI are selected because they are, including IBM Watson Studio, in the so-called *Leaders* quadrant of the Gartner Magic Quadrants 2021 for Cloud AI Developer Services[30].

4. ANALYSIS

4.1 Azure Machine Learning

Azure Machine Learning is Microsoft's platform and service for the life cycle of a machine learning project. In order to understand the deployment of a machine learning model on Azure Machine Learning, some terms have to be explained.

4.1.1 Explanation of terms

- An endpoint itself can have multiple **deployments**. A deployment is a specification of several resources and attributes that are necessary for hosting a model on a compute target. The attributes vary depending on the type of model and inferencing. The data traffic from inferencing can be divided in an endpoint using the concept **blue/green deployment**, which is the concept of diverting traffic between a blue deployment and a green deployment in order to change deployments gradually or to spread the compute load. This is possible since an endpoint can have multiple deployments.
- A **scoring script**, also called entry script, is needed. It receives the input data for inferencing, passes it to the deployed model and then returns the response of the model.
- An **environment** for the model which specifies what docker image or operating system is used, for example,

is needed. The docker image mentioned earlier is a container which ensures that software runs the same regardless of the computing system.

- A **compute target** is the system on which the machine learning model is hosted on. Some examples are Azure Kubernetes Service, Azure Container Instances or a local machine[36].

4.1.2 Deployment process (R1, R3, R4)

Azure Machine Learning can deploy various types of machine learning models, for example, for batch or online inference. In the following, the deployment to online and batch endpoints will be summarized as shown in figure 1.

The deployment of a model with an online endpoint on Azure CLI requires knowledge in Python, YAML and Azure Command Line Interface itself, also known as Azure CLI. Azure CLI and YAML do not have to be used if the deployment of the model is done on the Azure Machine Learning studio, which is an online portal for creating and managing machine learning projects, with a user interface.

- Firstly, Azure CLI v2 and its machine learning extension, an Azure resource group and an Azure workspace have to be already set up. An online endpoint for inferencing with its deployments is also needed.
- The next step is to register the model and environment separately using Azure CLI.
- In order to be able to send inference requests to the model, a scoring script has to be available.
- With the components consisting of a registered model, registered environment, an endpoint with at least one deployment and a scoring script, the model can be deployed. The deployment requires the user to create the endpoint and its deployments in the cloud by using Azure CLI.
- An inference can be requested by sending the request data through an Azure CLI command or by using a REST client.

If Azure Machine Learning studio is used, the model can be uploaded and registered from the studio. Furthermore, during the deployment process Azure Machine Learning Studio uses a setup wizard which creates an endpoint and the deployments. The environment and scoring script have to be uploaded in the following steps of the wizard[36].

The deployment of a batch inferencing model differs from an online inferencing model only in an additional step where the compute target, which is an Azure Machine Learning compute cluster, has to be created beforehand and in the content of the files[38].

The machine learning model and environment do not require to change to a format unique to Azure ML. The output of training jobs, such as the model directory which is output by MLflow or a scikit model in pkl format, can be used without any changes[37] [35].

Developers require knowledge about Azure CLI, YAML schema of endpoints, deployment and environment if the model is not deployed on Azure Machine Learning studio. Furthermore, a Python scoring script with an init() method and a run() method are needed. The run() method receives input data, passes the data to the model for inference and returns the inference result. In the init() method any initialization tasks such as caching the model into memory, can be executed.[36].

4.1.3 Explaining the process for machine learning model deployment on Azure Machine Learning

Based on the deployment process of models on Azure Machine Learning regardless of inference and compute target type five core components can be inferred as part of a structure as shown in figure 2. The components are the endpoint, the deployments of the endpoint, the scoring script, the machine learning model and the environment.

The following figure 2 describes only the flow of the inference request and does not reflect the authentication at the endpoint for example.

- At first, the inference request reaches the endpoint.
- The endpoint divides the traffic between all deployments according to its specification if there is more than one deployment.
- In the next step, the scoring script receives the inference request and forwards the to be inferenced data to the model.
- The model is running in the environment on the compute target and offers the scoring script a model response which is returned to the endpoint.
- Finally, the endpoint sends the model response to the sender of the inference request.

4.2 Amazon SageMaker

Amazon SageMaker is Amazon's platform and service for managing machine learning models including their deployment. The deployment of machine learning models on Amazon SageMaker is divided into four types of inference.

- **Real-time inference** like described in the Background.
- **Serverless inference** automatically manages and scales compute resources which is recommended for workloads that have idle periods between high traffic periods.[12]
- **Asynchronous Inference** is recommended for inference requests which have high data sizes up to one GB. The requests are queued and processed in an asynchronous manner which is possible with automatic scaling and management of computing resources.[2]
- **Batch transform** uses a machine learning model and a dataset to create a transform job which uses a real-time endpoint for batch inferencing[34].

4.2.1 Deployment process (R1, R3, R4)

Amazon SageMaker offers several options for the deployment of machine learning models. The SageMaker console has a user interface for an interactive deployment. It is also possible to use AWS CLI or AWS SDK's like Boto3 for deployment using code[6]. The deployment process (figure 3) is very similar between all inference types. It consists of four steps if the SageMaker Python SDK is used.

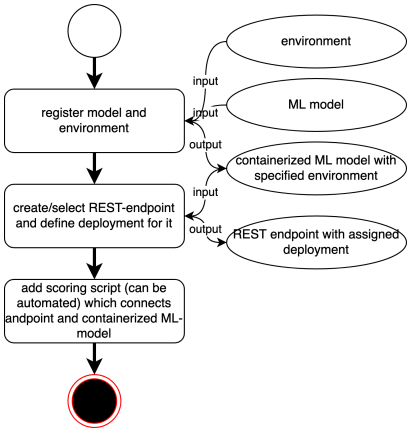

Figure 1: Graphical representation of the deployment process on Azure Machine Learning

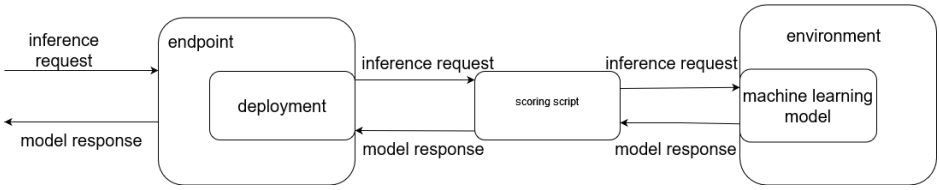

Figure 2: Graphical representation of the deployment of machine learning models in Azure Machine Learning

- At first the AWS region and SageMaker role, which defines which permissions the user gives SageMaker, have to be specified.
- After that, the container image has to be retrieved and the trained model has to be provided.
- Then a model object has to be created.
- Finally, the endpoint has to be created and the model has to be deployed[6][4][5].
- Requesting an inference requires a SageMaker Runtime client and SageMaker API has to be used[9].

Another concept which can be used for deployment is the blue/green deployment. It enables a smooth switch between deployments of, for example, different versions of a model. In order to make use of the concept, the endpoint can be updated[3]. The interactive approach in the Amazon SageMaker console offers a wizard to execute the previously mentioned steps[7].

SageMaker relies on Docker containers for the deployment of machine learning models and thus any model developed in a supported framework on SageMaker or with a pre-built Docker container has a fitting format and can be deployed. In the case of models which are serialized in JSON or pkl or without any Docker container, a container can be created with the ezsmdeploy SDK. It uses the Amazon SageMaker Python SDK and requires the model files, a python script for inference and prediction and a file which lists dependencies[40][8][15].

Thus, a developer requires knowledge of the Amazon SageMaker Python SDK, SageMaker API, Python and ezsmdeploy SDK or Docker containers for the deployment of machine learning models.

4.2.2 *Explaining the process for machine learning model deployment on Amazon SageMaker*

The previously introduced steps needed to deploy a model seemingly only require the container, model and an endpoint as components. The definition and documentation of these mentioned components offer a better insight into what components are needed for the deployment and which are optional.

Starting with the step of creating a model object in SageMaker, the model object can accept more parameters than the container, the model URL and the SageMaker role. The documentation lists a predictor_cls() which is method to create a **predictor**[10]. A predictor makes inference requests, encodes input data and decodes output data. The predictor encodes data with a serializer that is passed as a parameter to the predictor. In the same way, a deserializer can be passed to the predictor for decoding the output data. Since a predictor is only created if a method predictor_cls() is passed as a parameter to the model and the model calls its deploy() method, the predictor is optional[10] [11].

The last step of creating the endpoint and deploying the model requires the model object to call its deploy() method. The deploy() method receives the instance type, instance count and endpoint name as parameters and creates the endpoint on which the model is deployed. The endpoint itself can be updated as mentioned in the earlier paragraph. Using the Amazon SageMaker API in Boto3, the method update_endpoint() changes the already created endpoint by deleting the old endpoint configuration and deploying a new one. The endpoint configuration has a deployment configuration, which is an attribute which makes the blue/green deployment possible[13]. However, the definition of the endpoint configuration and deployment configuration is optional

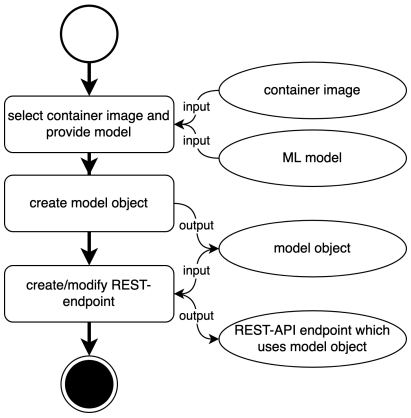

Figure 3: Graphical representation of the deployment process on Amazon's SageMaker

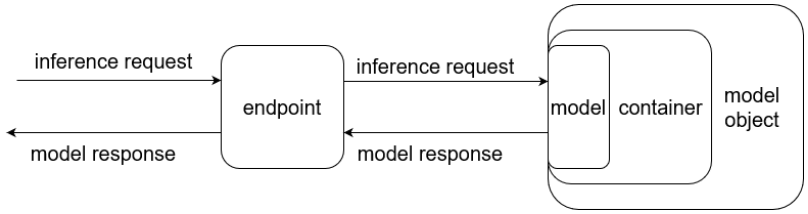

Figure 4: Graphical representation of the use of a deployed model on Amazon SageMaker

since they are not required in the deployment process.

Without the optional components, the deployment of a machine learning model can be represented as the following diagram 4. The endpoint receives inference requests and passes them to the model object. The model object which has the container and machine learning model sends, after inferencing, the model response to the endpoint.

4.3 Google Cloud's Vertex AI

Google unified its preceding AI Platforms into the new Vertex AI platform on May 18th, 2021 [42]. The platform allows users to create machine learning models and AI applications within a no-code environment (AutoML) or custom-trained models based on arbitrary coding (AI Platform) [24]. Google offers to deploy models created within their infrastructure as well as models created and trained on other platforms.

4.3.1 Deployment process (R1, R4)

Independent of the used scenario, which are custom-trained models or the use of Google's AutoML, the user needs to create a REST endpoint within the Google Cloud. Google enables the user to deploy multiple models to the same node, e.g. for gradually version changes in a model or load balancing. It is also possible to deploy one model to different endpoints, e.g. testing and production environment or usage of the same model with different performance needs [19]. A visualization of the process can be found in figure 5.

Google's Cloud Platform offers deployment of custom software as well. Considering this, every type of machine learning software can be deployed. To make use of the beneficial features of Vertex AI, e.g. using pre-trained containers, using containerization on Google servers and their optimizations considering speed and latency, the model needs to be imported, if not already existing, to Vertex AI. This section is focused on this part because especially here Google defines its processes for how to deploy machine learning models.

Regarding research question R3: For models that are created within special Python frameworks (TensorFlow, scikit-learn, XGBoost) and versions, at least Python 3.7, the user can work on pre-configured containers provided by Google [28]. Users that would like to use models created outside the Google Platform need to import their code as images to either Googles' Artifact Registry [26] or Googles' Container Registry [20]. These containers can then be used within Vertex AI [29].

Once a model is created within Vertex AI users can make use of the versioning and life cycle management of their models [27].

Google offers deployment in two versions: Creating an online, synchronous prediction service or accessing an asynchronous batch processing. The processes are different for both types and eventually introduce restrictions.

Online prediction.

Models from the Vertex AI suite can be deployed either with the dedicated Vertex AI API [22] or with the more general Google Cloud Console [21]. Both variants enable the user to leverage blue/green deployment by assigning an improved model to an already existing REST endpoint and adjust the so-called traffic split between these models. The user can make use of load balancing and horizontal scaling due to replication of the containers used. A relevant restriction for online prediction is the unavailability of online prediction for models based on Google's AutoML Video service. For using the endpoint, a user makes synchronous HTTP calls to the created REST endpoint in specified formats. These formats vary among the AutoML services and

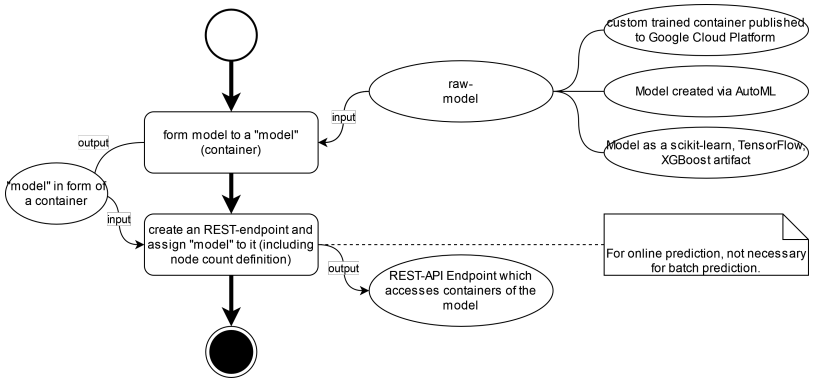

Figure 5: Graphical representation of the deployment process on Google Cloud Platforms Vertex AI

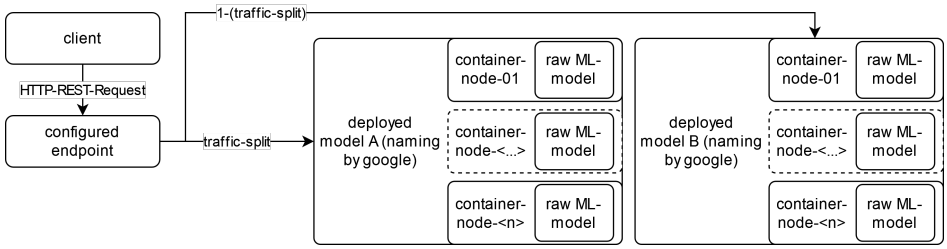

Figure 6: Graphical representation of the use of a deployed model on Google Cloud Platforms Vertex AI

especially if a custom-trained or imported container is used. In figure 6 a visualization of the interaction with a deployed model is given.

Batch predictions.

The steps described in the paragraph "Online prediction" are not relevant if a user would like to make batch predictions. Within Vertex AI a model, which is used to make batch predictions, does not need to be deployed to an endpoint. The user can specify a batch prediction service with defined input and output destinations. To specify the model and its version is relevant as well. To trigger a batch process, the user uses the Vertex AI API. For integrating this service into a larger software, the software has to communicate via the Vertex AI API and may need to store and load the input and output data programmatic to the Google Cloud [23]. The differences compared to online-prediction can be seen in figure 6 as well.

Required Developer Knowledge (R4).

For deploying an already created model within VertexAI a user needs to be familiar with the concept of REST-APIs and able to make use of the Google Cloud API. If it is necessary to create containers from given programs Docker knowledge is also required. In general, the user does not have to be a skilled developer but has to be familiar with special information technology concepts like REST or Containerization.

4.3.2 What are the steps and components of a machine learning model deployment on Google Vertex AI?

The deployment can be split into three major components and related steps:

- raw **model** as code or as python artifact or as AutoML artifact
- **container** created by Vertex AI from python artefact or AutoML artefact or imported by the user. *Google calls this model which should not be confused with the raw model.*
- **endpoint** which is configured to horizontal scale or use special resources, e.g. GPU, and direct traffic to containers, so-called models

4.4 Comparison (R2)

4.4.1 Deployment process results

Online-processing services.

The final result of the machine learning model deployment processes is very similar among the different platforms. Every platform deploys for online predictions to **REST-API endpoints** which are then used by clients. All platforms use **containers** for deployment and offer a **horizontal scaling** via duplicating container instances. Every platform allows some type of **blue/green deployment** with different techniques.

Batch processing services.

Utilizing batch processing services differ among the analyzed platforms. Azure ML specifies special batch processing endpoints. Googles' Vertex AI allows to create batch processing jobs on containers, which are called models, but for utilizing them in an application manual coding is needed. Amazon SageMaker uses online processing endpoints with special parameters.

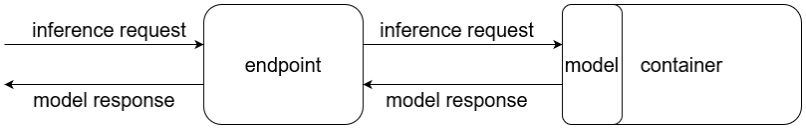

Figure 7: General deployment pattern of the three analyzed tools

4.4.2 Deployment process (R2)

Components and developer provided inputs.

Every platform needs a **model** which should be deployed. The form in which a model is available can vary within every platform. Amazon's SageMaker and Googles' Vertex AI create a container image from a model which is presented as an output of their own MLOps services or as an artefact created from other software, e.g. TensorFlow, scikit-learn or XGBoost. It is also possible for every platform to deploy an already created container. Using arbitrary containers in the process is quite similar to a general deployment process but may lead to non-optimal use of resources as the containers created by the platforms themselves make use of special configurations for ML-related algorithms.

Only Azure ML needs to have a so-called *scoring-script* which is used as middleware between a REST endpoint and a containerized model. It is possible to specify the environment in more detail for every platform.

Core deployment processes.

The first step of all processes is the derivation of a **containerized** version of an already created machine learning model, whose format may vary. For every platform, the developer needs to create an **endpoint** which handles the requests and communicates with several instances of the previously described containers. All analyzed platforms enable a **blue/green deployment** by allowing to use multiple models in form of different containers behind a single endpoint. The developers or administrators can define the amount of traffic which is sent to which model.

Deployment of batch-processing services.

For batch predictions, the processes vary. Azure ML uses a quite similar process compared to its online-prediction deployment process. An endpoint can be either batch processing or online processing. Amazon SageMaker utilizes online endpoints for batch processing with specialized parameters. Googles' Vertex AI needs a little less configuration for batch-processing as it is not needed to create an endpoint but it is possible to define batch processes within the API or the UI of Vertex AI. The developer needs to add data transfers and API-Calls to its application or service which should use batch prediction manually. If it is only needed on a server-side level a type of cronjob can be defined.

4.4.3 Developer skills and operation of the platform's services (R4)

If a containerized version of a model is already present, every platform enables low coding to no coding deployment only using a dedicated API or a Web UI. Only Azure ML requires the user who deploys a model to write the scoring script. For all platforms, a developer should be familiar with the concepts of containerization and REST APIs.

4.4.4 Pattern in deployment processes

From all the similarities and differences it is possible to derive a general pattern which can be found with small deviations in every analyzed platform 7.

At first, every platform forces the developers to create a containerized version of their machine learning model. The developer must create a REST endpoint, within the platform which communicates with one or more machine learning models, as well. Every platform utilizes the containers for horizontal scaling which is parameterized by the developer. For updating a model every platform presents a blue/green deployment technique.

5. CONCLUSION

In summary, Azure Machine Learning requires a model, a scoring script, an endpoint, at least one deployment and an environment for the deployment of a machine learning model. In comparison, Amazon SageMaker needs an endpoint and a model object made of the machine learning model and container for deployment. Lastly, Google's Vertex AI deploys a model only with a model, an endpoint and a container. Azure Machine Learning has the most differences from the other two tools since a scoring script and at least one deployment are also necessary for Azure Machine Learning.

Therefore, the general deployment pattern of the three analyzed platforms is made up of the following common components:

- A machine learning model,
- an endpoint
- and a container or some sort of another environment.

While the found pattern is not representative of all MLOps platforms, since only three platforms have been analyzed for the pattern, it indicates a potential trend in how MLOps platforms deploy machine learning models in the future.

Furthermore, the results show that, at least concerning three of the currently leading, according to Gartner[30], platforms, MLOps platforms are becoming more similar to each other.

6. FUTURE WORK

This paper only offers a comparison of the deployment process on three MLOps platforms. To decide with certainty whether a standard is forming, other platforms have to be analyzed as well in regards to their deployment process and their required components for deployment. Furthermore, in the case a standard can be inferred, a general deployment pattern can be defined such as in this case.

7. REFERENCES

[1] Amazon. Amazon SageMaker. https://aws.amazon.com/sagemaker/?nc1=h_ls. 28.05.2022.

[2] Amazon. Asynchronous inference. https://docs.aws.amazon.com/sagemaker/latest/dg/async-inference.html. 28.05.2022.

[3] Amazon. Blue/Green Deployments. https://docs.aws.amazon.com/sagemaker/latest/dg/deployment-guardrails-blue-green.html. 28.05.2022.

[4] Amazon. Create a serverless endpoint. https://docs.aws.amazon.com/sagemaker/latest/dg/serverless-endpoints-create.html. 28.05.2022.

[5] Amazon. Create an Asynchronous Inference Endpoint. https://docs.aws.amazon.com/sagemaker/latest/dg/async-inference-create-invoke-update-delete.html. 28.05.2022.

[6] Amazon. Create your endpoint and deploy your model. https://docs.aws.amazon.com/sagemaker/latest/dg/realtime-endpoints-deployment.html. 28.05.2022.

[7] Amazon. Deploy a Compiled Model Using the Console. https://docs.aws.amazon.com/sagemaker/latest/dg/neo-deployment-hosting-services-console.html. 28.05.2022.

[8] Amazon. Frameworks. https://sagemaker.readthedocs.io/en/stable/frameworks/index.html. 13.06.2022.

[9] Amazon. InvokeEndpoint. https://docs.aws.amazon.com/sagemaker/latest/APIReference/API_runtime_InvokeEndpoint.html#API_runtime_InvokeEndpoint_RequestParameters. 28.05.2022.

[10] Amazon. Model. https://sagemaker.readthedocs.io/en/stable/api/inference/model.html?highlight=model. 28.05.2022.

[11] Amazon. Predictor. https://sagemaker.readthedocs.io/en/stable/api/inference/predictors.html. 28.05.2022.

[12] Amazon. Serverless Inference. https://docs.aws.amazon.com/sagemaker/latest/dg/serverless-endpoints.html. 28.05.2022.

[13] Amazon. UpdateEndpoint. https://docs.aws.amazon.com/sagemaker/latest/APIReference/API_UpdateEndpoint.html. 28.05.2022.

[14] Z. Arnold and H. Toner. Deep learning for recommender systems: A Netflix case study. *AI Magazine*, 42(3):7–18, 2021. https://doi.org/10.1609/aimag.v42i3.18140.

[15] M. Du and T. Hughes. Train and host Scikit-Learn models in Amazon SageMaker by building a Scikit Docker container. https://aws.amazon.com/de/blogs/machine-learning/train-and-host-scikit-learn-models-in-amazon-sagemaker-by-building-a-scikit-docker-container/. 13.06.2022.

[16] I. Gartner. Gartner identifies the top strategic technology trends for 2021. https://www.gartner.com/en/newsroom/press-releases/2020-10-19-gartner-identifies-the-top-strategic-technology-trends-for-2021#:~:text=Gartner%20research%20shows%20only%2053,a%20production%2Dgrade%20AI%20pipeline. 19.10.2020.

[17] I. Gartner. Gartner survey reveals 66% of organizations increased or did not change ai investments since the onset of covid-19. https://www.gartner.com/en/newsroom/press-releases/2020-10-01-gartner-survey-revels-66-percent-of-orgnizations-increased-or-did-not-change-ai-investments-since-the-onset-of-covid-19. 01.10.2020.

[18] J. Halstenberg, B. Pfitzinger, and T. Jestädt. *DevOps Ein Überblick*. essentials. Springer Vieweg Wiesbaden, 1. edition, 2020.

[19] G. I. Limited. Considerations for deploying models. https://cloud.google.com/vertex-ai/docs/general/deployment. 26.05.2022.

[20] G. I. Limited. Container registry overview. https://cloud.google.com/container-registry/docs/overview. 26.05.2022.

[21] G. I. Limited. Deploy a model using the cloud console. https://cloud.google.com/vertex-ai/docs/predictions/deploy-model-console. 26.05.2022.

[22] G. I. Limited. Deploy a model using the vertex ai api. https://cloud.google.com/vertex-ai/docs/predictions/deploy-model-api. 27.05.2022.

[23] G. I. Limited. Get batch predictions. https://cloud.google.com/vertex-ai/docs/predictions/batch-predictions. 27.05.2022.

[24] G. I. Limited. Introduction to vertex ai. https://cloud.google.com/vertex-ai/docs/start/introduction-unified-platform. 26.05.2022.

[25] G. I. Limited. MLOps: Continuous delivery and automation pipelines in machine learning. https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning. 28.05.2022.

[26] G. I. Limited. Overview of artifact registry. https://cloud.google.com/artifact-registry/docs/overview. 26.05.2022.

[27] G. I. Limited. Overview of artifact registry. https://cloud.google.com/vertex-ai/docs/model-registry/introduction. 26.05.2022.

[28] G. I. Limited. Pre-built containers for prediction and explanation. https://cloud.google.com/vertex-ai/docs/predictions/pre-built-containers. 26.05.2022.

[29] G. I. Limited. Use a custom container for prediction. https://cloud.google.com/vertex-ai/docs/predictions/use-custom-container. 27.05.2022.

[30] P. Mechliński. Use a custom container for prediction. https://www.linkedin.com/pulse/top-5-ai-cloud-providers-ranking-2021-piotr-mechli%C5%84ski. 21.09.2021.

[31] Microsoft. Azure machine learning. https://azure.microsoft.com/en-gb/services/machine-learning/#product-overview.

24.05.2022.

[32] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of Machine Learning*. Adaptive Computation and Machine Learning Ser. MIT Press, August 2012.

[33] P. Ponnusamy, A. R. Ghias, Y. Yi, B. Yao, C. Guo, and R. Sarikaya. Feedback-Based Self-Learning in Large-Scale Conversational AI Agents. *AI Magazine*, 42(4):43–56, 2022. https://doi.org/10.1609/aimag.v42i4.15102.

[34] see contributors. Amazon SageMaker Batch Transform. https://github.com/aws/amazon-sagemaker-examples/blob/main/sagemaker_batch_transform/introduction_to_batch_transform/batch_transform_pca_dbscan_movie_clusters.ipynb. 28.05.2022.

[35] see contributors. Azure. https://github.com/Azure/azureml-examples/tree/main/cli/endpoints/online/mlflow. 07.06.2022.

[36] see contributors. Deploy and score a machine learning model by using an online endpoint. https://docs.microsoft.com/en-gb/azure/machine-learning/how-to-deploy-managed-online-endpoints. 24.05.2022.

[37] see contributors. MLflow Models. https://www.mlflow.org/docs/latest/models.html. 07.06.2022.

[38] see contributors. Use batch endpoints for batch scoring. https://docs.microsoft.com/en-gb/azure/machine-learning/how-to-use-batch-endpoint. 24.05.2022.

[39] H. Steck, L. Baltrunas, E. Elahi, D. Liang, Y. Raimond, and J. Basilico. AI Accidents: An Emerging Threat. *Center for Security and Emerging Technology*, 2021. https://doi.org/10.51593/20200072.

[40] S. Subramanian. Deploy machine learning models to Amazon SageMaker using the ezsmdeploy Python package and a few lines of code. https://aws.amazon.com/de/blogs/opensource/deploy-machine-learning-models-to-amazon-sagemaker-using-the-ezsmdeploy-python-package-and-a-few-lines-of-code/. 13.06.2022.

[41] M. Treveil, N. Omont, C. Stenac, K. Lefèvre, P. Du, and M. Fraaß. *MLOps – Kernkonzepte im Überblick Machine-Learning-Prozesse im Unternehmen nachhaltig automatisieren und skalieren* . Animals. O'Reilly Verlag, Deutsche Ausgabe edition, 2021.

[42] C. Wiley. Google cloud unveils vertex ai, one platform, every ml tool you need. https://cloud.google.com/blog/products/ai-machine-learning/google-cloud-launches-vertex-ai-unified-platform-for-mlops. 18.05.2021.

Anatomy of a Machine Learning Pipeline

David Abdelmalek
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
david.abdelmalek@rwth-aachen.de

Devashish Gaikwad
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
devashish.gaikwad@rwth-aachen.de

ABSTRACT

Machine learning is undergoing a revolution with an increase in the use of ML models in production. The goal of MLOps is to automate prototyping and deployments for rapid development. Many software companies have created their MLOps solutions by giving their interpretation of an ideal machine learning pipeline. But there is no common interpretation of how a generic machine learning pipeline should look and function. This paper provides a formalization of popular MLOps pipeline structures, intending to define a common anatomy of MLOps pipelines. This anatomy will provide a common terminology for communication about MLOps pipelines and help improve the modeling process.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*productivity, programming teams, software configuration management*

Keywords

Microsoft Azure, Machine learning, AWS, Spark, MLOps, Anatomy, Pipeline

1. INTRODUCTION

Over the last two decades, machine learning is encountered often in our daily lives in social media, movie recommendations [20], etc., due to significant data availability and substantial computing resources offered by various frameworks for managing the machine learning lifecycle. However, applying machine learning algorithms on a big scale comes with risks and challenges, considering auditing, stability, and software continuous delivery. Continuous delivery (CD) is a software development procedure to automate the release of a new version of code to production. As a result, a unified approach is required in the entire software lifecycle, starting from business requirements to end-product, going through the building, testing, and production, which is called MLOps. MLOps are a set of approaches focused on creating and deploying quality AI models in production efficiently and automatically. In MLOps, machine learning models can be tested and deployed in isolated environments, increasing production pace.

MLOps is a set of practices aiming to produce machine learning end-to-end products collaboratively between data scientists and ML engineers. Accordingly, MLOps has a proven set of advantages [23], for instance, *Reproducibility* by defining reusable functions for data preprocessing, training, and scoring process, *Versioning* by allowing developers to work on pipeline concurrently, *Reusability* by creating pipeline implementations which can be triggered externally using REST APIs, and much more.

Certainly, advantages can not be granted without having some concrete conventions to supervise stable workflow. Those principles ensure higher quality, simplify the management process, and continuous model delivery using continuous integration and continuous delivery software methods. Continuous Integration (CI) is a software development procedure that builds, tests, and merges new code changes to a repository. Continuous Delivery (CD) aims to automate the release of the latest version of code to production. Fig 1 presents 4 essential principles for independent MLOps workflows [23] and will be discussed subsequently.

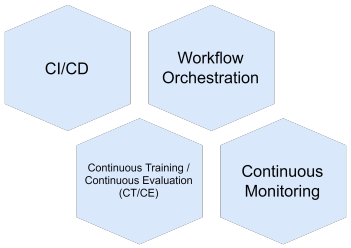

Figure 1: MLOps principals.

- **CI/CD** is a set of practices/techniques. One practice is to implement a delivery process as an automated delivery pipeline.
- **Workflow Orchestration** controls the pipeline stages synchronization using directed acyclic graphs (DAGs), defining task dependencies and relationships [23].
- **Continuous Training / Continuous Evaluation (CT/CE)** establishes automated training and evaluation on new

SWC Seminar 2018/19 RWTH Aachen University, Germany.

data after extracting feature sets from the data prepossessing stage.

- **Continuous Monitoring** monitors the model's behavior after deployment to ensure it works as expected and evaluates its performance based on a group of metrics.

Multiple offerings currently exist in the market encompassing the above principles. There is no common framework or anatomy which can describe these MLOps pipelines for further study. Therefore, the current solutions are reviewed, and then their anatomies are formalized into a generic one so that the understanding of the MLOps pipelines can be made manageable and optimizable in the future. This paper aggregates pipelines from different MLOps platforms and then provides a generalized MLOps pipeline with generic anatomy.

The structure of the paper will go as follows. First, the end-to-end MLOps pipeline in popular cloud providers and data processing frameworks - Azure, Spark, AWS, GCP, and Databricks is reviewed. After studying different pipeline structures, a formalized and generalized overview of the machine learning pipelines is given. Finally, the findings are concluded with a summary and proposed future work.

2. MLOPS PROVIDERS

2.1 Azure

Azure is a public cloud computing platform managed by Microsoft. Azure Machine learning is one of the services provided for building, managing, and monitoring machine learning models' lifecycle. Azure Machine Learning Studio is a web-based platform where data scientists and engineers find tools to create and automate high-quality day-to-day workflows.

In Azure, pipeline stages are encapsulated as several steps with different computing resources for each stage [24]. Hence, separate stages allow developers working on the same pipeline do changes to scripts independently without overloading pipeline computation. Fig 2 shows essential steps needed to employ an independent executable pipeline in Azure. In the next paragraphs, we will go through each of those stages.

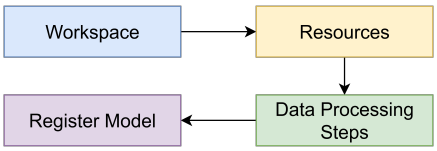

Figure 2: Azure ML workflow.

Step 1: Create workspace - a high-level resource for machine learning functions to manage and monitor created artifacts in a centralized platform - to hold and share pipeline [8].

Step 2: Setup pipeline resources (computation and datastore).

Step 3: Create machine learning model steps. The machine learning model in Azure is defined as separate steps. A step - a pre-built function available via the Azure Machine Learning SDK [27] - takes attributes such as script path, input, output, etc., and runs on the configured compute target resources. Each step executes a script, from the script directory, on the input source and stores results to the output datastore given as value. For instance:

- **Data Preparation**: This step first takes input data in any format and then operates different predefined data preprocessing modules to prepare/clean data for the training step. For example, selecting the clipping module to treat incorrect records, using the Normalize Data module to normalize input data, employing the Group Data Into Bins tool to prepare numerical data in addition to converting numeric features into categorical ones.

- **Training Evaluating**: Model training can be done in the Azure AutoML tool [10, 11]. Automated machine learning expeditiously computes different combinations of algorithms and hyperparameters, tracks the metrics' outcome from those models, evaluates models, and finally selects a model with the best results of the metric chosen.

Stage 4: *Model Register* is the last step for both deploying ML models and tracking model different versions. Each model is created as a docker container with all files storing the model. After registering, a model is identified by name and version, while each version is incremented once a model registers with the same name. For deployment, only registered models can be deployed as service endpoints. Encapsulated environment with all dependencies, scoring code to predict given request, and inference configuration are all required for successful deployment.

2.2 Spark

Apache Spark [1] is an open-source data processing platform used to process queries on big data and allocate data queries across multiple clusters. Spark can be deployed in various programming languages, Python, Java, and Scala, and provide code reuse for numerous workloads. By using distributed workloads advantage, Apache defines a machine learning pipeline that manages machine learning algorithms along with big data processing and provides an API (MLlib) that simplifies the development and deployment of scalable pipelines.

MLlib [1] - a spark machine learning library - is API that combines multiple powerful machine learning algorithms into a single workflow. In the next paragraphs, we will define the main component of the spark machine learning pipeline as depicted in 3

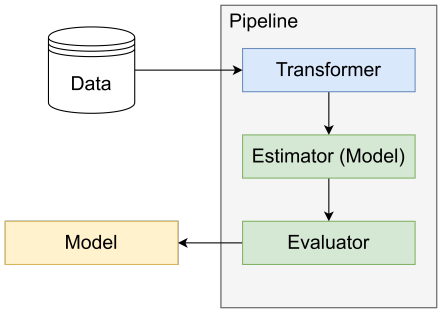

Figure 3: Spark ML workflow.

Data: MLlip API uses DataFrame from Spark SQL as a dataset uniform data structure, which can hold different data types, for instance, text, feature vectors, booleans, etc.

Transformer: is a one-level abstraction in MLlip API that acts as a preprocessing step containing learned models (output from estimators) and different feature transformation phases such as one-hot encoding and feature scaling. This stage technically executes *transform()* function which transforms one dataframe into another by adding, deleting, or updating features in the input dataframe and providing the new dataframe as input for the next stage.

Estimator: is another high-level abstraction that encapsulates algorithms that train/fits data. In the essence of training, the Estimator technically implements method *fit()*, which accepts transformed dataframe from the preprocessing step, trains data on ML algorithm, and returns a transformer. The returned transformer alters the dataframe according to parameters learned during the learning phase.

Evalulator: is the last step before returning model. It is used to evaluate model performance using ROC [21] and help with model tuning process by selecting the best model for generating predictions.

Model: Finally, the pipeline outputs a model with *fit()* method, which can be used to start the training of different models.

2.3 AWS

Amazon Web Services (AWS) is one of most most widely used Cloud Service Providers. AWS SageMaker Pipeline [6] is their continuous integration and continuous delivery (CI/CD) service for machine learning. SageMaker Pipeline builds upon already existing SageMaker services and are orchestrated according to the MLOps templates which list out the underlying resources needed.

In general, AWS SageMaker pipelines are created according to a default MLOps template and a pipeline structure defined using python SDK. SageMaker project templates offer different choices of code repositories, workflow automation tools, and pipeline stages. Pipelines can be categorised into 3 stages with human checks requirements as per specifications. Fig 4 shows general structure of the pipeline. In next paragraphs we go through the each stage and general steps involved in them.

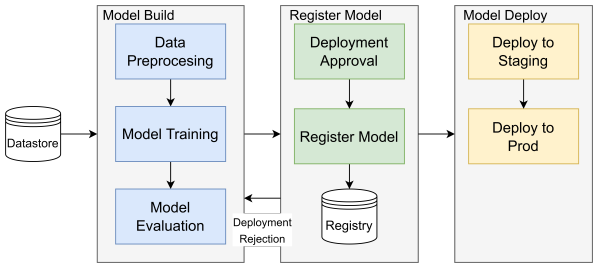

Figure 4: Machine learning workflow in AWS

Model Build: Model building stage focuses on the data pre-processing, training and tuning of the model.

- Processing: Processing Jobs are created for data preprocessing. A processing step requires a processor component (for example, SKLearnProcessor from SKLearn [13]), python script which defines the sequence of actions, data read and write locations (S3 object storage, for example).

- Training: Training jobs are created to train a model. A training step requires an estimator, as well as training and validation data inputs. Here, user can select from a plethora of built-in algorithms and pre-trained models like XGBoost [14], K-NN, YOLO [30], etc. as well as define custom models.

- Tuning: Tuning jobs are created to tune hyperparameters, also known as hyperparameter optimization (HPO) [29]. It runs multiple training jobs, each one producing a model version. Tuning jobs produce model artifacts such as model weights and hyperparameters.

- Transform: Batch transform the input dataset by running inference on it. Outputs a dataset. it also allows to optimize the job by specifying different parallelization parameters for any pre-processing.

- ClarifyCheck: ClarifyCheck conducts baseline drift checks against previous baselines for bias analysis and model explainability. Checks such as data bias check, model bias check and model explainability check can be done one at a time. Outputs results.

Register Model: The Model is registered in the AWS SageMaker Registry for deployment and metrics comparison.

- RegisterModel: RegisterModel registers the created model with AWS SageMaker model registry, each model can be one model or a combination of two or more models. Model infrastructure requirements as well any baselines are also registered. Approval by human or automatic approval can be programmed in the step.

Model Deployment: MLOps project templates include model deployment specifications, approved model versions in the model registry are automatically deployed to production. Desired endpoint, infrastructure and API specifications have to be set and deployed using the CreateModel jobs.

Helper Steps: Certain steps are used for flow and logic control.

- Condition: Condition Steps are used to evaluate the specified conditionals to assess which action should be taken next in the pipeline. Nested condition steps are currently unsupported.

- Callback: Callback steps incorporate additional processes and AWS services into the workflow that aren't directly provided by Amazon SageMaker Model Building Pipelines.

- Amazon Elastic Map Reduce: Runs Map Reduce functions [16] on the dataset.

- Fail: Fail steps are run when a desired condition or state is not achieved and we want to mark the execution of pipeline as failed, along with custom error messages.

2.4 GCP

Tensorflow Extended (TFX) [26] - is the MLOps framework used by Google Cloud Platform (GCP). Under the hood TFX supports several orchestrators such as Kubeflow, apache Airflow and Apache Beam. In certain installations of TFX, there is no need for user to specify and manage any of the underlying infrastructure. TFX works by specifying pipelines files beforehand. Most common way of the pipeline building and scripting for TFX is done using an online jupyter notebook provided by Google cloud.

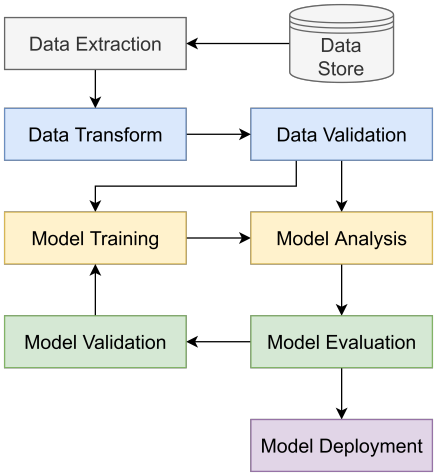

Figure 5: Machine learning workflow in GCP using TFX

TFX pipelines consist of several components which are mentioned below.

Data Preprocessing

- **ExampleGen**: ExampleGen splits the data into training and evaluation datasets. It accepts input data and creates data as output in different formats such as CSV, TFRecord, BigQuery, etc
- **StatisticsGen**: StatisticsGen calculates the dataset statistics like min, max, distribution, missing values etc by consuming the datasets created by ExampleGen and creating dataset statistics.
- **SchemaGen**: SchemaGen creates a data schema showing the expected type of data for each feature by consuming statistics from StatisticsGen and creating Data Schema.
- **ExampleValidator**: ExampleValidator detects anomalies, missing values and data skew by consuming schema and statistics from SchemaGen and StatisticsGen respectively and generating data validation results.
- **Transform**: Transform does feature engineering on the data created by ExampleGen using a schema created by SchemaGen. It creates a model for feature engineering the data along with statistics. Transform enriches the dataset. It consumes examples (dataset) and dataset schema and produces a model which is stored to disk for later usage with any tensorflow compatible framework.

Model Building and Training

- **Trainer**: Trainer trains the model. it creates the models and variables related to them. It also creates a graph specific to the TensorFlow framework which stores global information, model and variable data, offering robustness, reliability and visualization of processes. We specify where, how and the architecture of the model. It consumes dataset (examples), trainer logic file, data schema, pre-trained models if any and creates atleast one inference model and another model for evaluation.
- **Tuner**: Tuner tunes the model's hyperparameters. It is primarily used with Keras components but can also be imported from outside libraries. Tuner consumes dataset, model and objective metrics to produce the optimum hyperparameters.

Model Validation

- **Evaluator**: Evaluator evaluates the performance of the model. It also validates the model using methods such as K-Means validation and calculating metrics such as AUC, loss, etc and comparing them with the baselines set by the devloper. It consumes evaluation split from ExampleGen and a trained model from Trainer. It "blesses" models which pass the evaluation tests, that is, the models are marked as good. It also produces analysis and validation metadata.

Model Deployment

- **Pusher**: Pusher pushes a model to deployment. We specify the endpoints and respective infrastructure here. It consumes a "blessed" (see *Evaluator* above) model and uses Tensorflow Serving to deploy the model.

2.5 Databricks

Databricks [2] is a data engineering and machine learning open-source platform for processing and transforming big data from various cloud-based data providers, ie. Azure, AWS, GCP, etc., to extract BI reports and build machine learning models. Moreover, Databricks is a web-based data warehouse for many data requirements. It is considered the primary service used by data-driven decision-making companies as a big data tool to achieve the full potential of combining their data efficiently, ETL processes, and machine learning. Designed by the founder of Apache Spark [31], Databricks offers a platform as a service (PaaS) on the top of cloud computing providers as which they manage resources (upscaling and downscaling computing clusters) based on workload.

Built upon centralized datalake, Databricks machine learning [4] provides an integrated complete ML lifecycle environment. It incorporates managed services for access data at any scale, feature management, model training and testing, and model serving. There are distinct machine learning architecture models in Databricks, but fig 6 depicts the most generic ML pipeline. We will establish a complete definition of each phase inside diagram in the following paragraphs:

First and foremost, data needs to be loaded into the ML pipeline. Databricks facilitates data ingestion procedure by having automated and reliable ETL functions from secure cloud data storage into centralized *Delta Lake*.

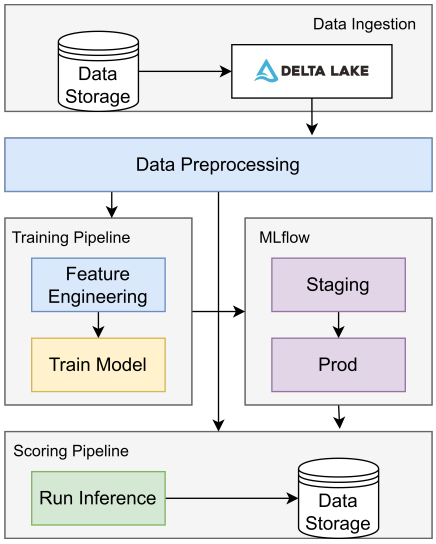

Figure 6: Databricks ML workflow.

Coming from different secure cloud data providers, Data in Data Lake is in its native format; hence, it requires selective **Data Prepossessing** steps such as handling null values, standardization/normalization, categorical variables, one-hot encoding, etc. Transformed output data can be easily now interpreted by ML models.

Data is now ready for the next phase, the **Training pipeline**. It has two interchangeably components which are:

- **Feature engineering**: takes raw input dataset and is responsible for not only extracting beneficial features that influence model prediction scores but also operating complex calculations such as PCA [19], word embedding [7], etc. A pipeline can be created manually by the developer or by *Feature store* [3]. Databricks Feature Store can create, explore and select features in the dataset and then publish features to a generalized repository across the organization to ensure computing the same feature code in model training and testing.

- **Train model** can be done either manually or using AutoML. All through preparing the dataset for model training until having the robust best-performed model across multiple ML models concerning development-based parameters and production-based parameters, *AutoML* can create machine learning models in regression, classification, and forecasting problems. MLFlow performs a set of training trails while evaluating various models from scikit-learn [28] and XGBoost [15] packages to select the best model. AutoML uses Spark and Hyperopt [9] - an open-source library that uses Bayesian optimization for parameter tuning- to paralyze the search for the best model across different models. Finally, it creates a python notebook with source code executed and trial logs for each run.

Next step is deploying the best model from the training pipeline as a REST API service using MLflow. MLflow Model Registry **MLflow**- an open-source service used for monitoring and governing machine learning's end-to-end lifecycle - uses the selected model artifact stored from AutoML to migrate from a new model version to staging and then production stages.

Scoring pipeline runs the model scoring script. First, the notebook loads the testing dataset (data with no predicted label(s)) from the data prepossessing stage to ensure reliable data format as well as the production version of the registered ML model. Then the trained model classifies input data and outputs some predictions (scores). Consequently, predictions along with loaded data directly move to one of the secure cloud data providers. In addition, *Auto Loader* trigger function automates the scoring pipeline by creating a streaming job once it detects a new input record and passes it to the score function.

3. GENERALIZATION

3.1 Formalization Proposal

As seen in the study section 2 Different major MLOps providers structure their MLOps pipelines differently with different components. There is no common definition of a MLOps pipeline. Different MLOps maturity level pyramids from Google [5] and Microsoft [18] try to classify the current state of MLOps deployments, but none of them generalize the common stages in a MLOps pipeline. To understand the common model of MLOps we first need to generalize across major MLOps providers to find out the commonalities and formalized stage descriptions.

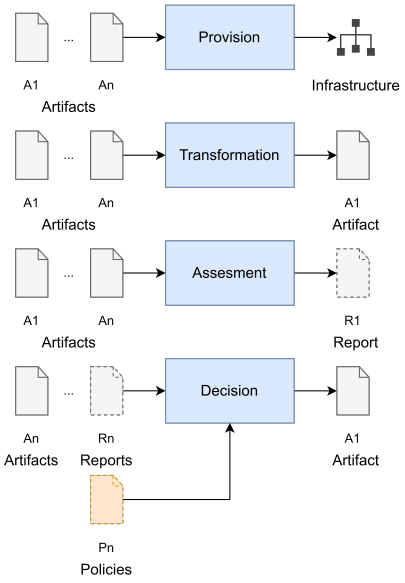

Figure 7: Generalization of Activities in MLOps

From the study in 2, we can generalize the types of activities and objects for any MLOps system. This provides us with the basic building blocks of any MLOps stage. All the activities shown in Fig. 7 can be combined together in different ways and for different number of times to create a single stage.

3.2 Artifacts

Artifacts are the static components of any pipeline. They can be data or physical infrastructure generated by users and

machines. Activities work on the artifacts to change them or create new ones. Artifacts act as an input or output to activities.

- **Model Artifacts**: Model artifacts are machine learning models, model specifications and meta data about the models.
- **Data Artifacts**: Data artifacts are datasets and data storages.
- **Script Artifacts**: Script artifacts are code, scripts and pipeline templates.
- **Reports**: Reports are the output of assessment activities. They contain model performance metrics, model validation reports, model evaluation metrics and infrastructure performance metrics. Any output which is a human readable measurement and summary of an artifact is a Report.
- **Policies**: Policies are the logics which guide the decision processes. They are generally set by the architect and direct the decision processes for the pipeline branches, if any. These consist of conditional statements based on the model evaluation metrics to decide whether more tuning is required or if the model should be deployed.
- **Infrastructure**: The underlying cloud infrastructure resources for hosting the MLOps activities and reading, storing objects. These can be physical (servers, databases, etc) or virtual (Workspaces, registries, extra-pipeline operations etc)

3.3 Activities

Activities are the process steps in any MLOps pipeline. They process the artifacts. One activity or more than one activities are stacked together create stages in any pipeline.

3.3.1 Transformation

Transformations are the main class of activities. Transformation work on one or more artifacts to produce another artifact. For example - Training a model, applying the transform operations on the datasets, etc. Following are the important transformation activities:

- **Data Processing**: Data processing transformation activities are related to reading data from the disk and processing them in one way or another to make them ingestable by the models. This also includes increasing the quality of the data and feature extraction. Generally, the input and outputs are dataset artifacts.
  1. **Data Retrieval**: These transformation read the data from physical storages or API endpoints into datasets which are then processed further. DataFrame Reader in Spark and built-in Bucket or Object storage reading functionalities in Azure and AWS are good examples.
  2. **Data Analysis**: These increase the quality of data by removing or fixing missing values and doing a statistical analysis of the data to look for data skew [25] which induce bias in the models. Data Correction in Azure and ExampleValidator in GCP are some notable examples. Datasets are analysed to create reports and metrics about the data.
  3. **Data Preprocessing**: These apply mathematical or transformative functions on the existing data to change its distribution and make it easier for the models to fit the data. It is also possible to change the data representation by creating categorical values from continuous ranges or changing textual data to word embeddings such as Fasttext [12], etc. Data Normalization, Binning, Categorization activities in Azure, Feature scaling, Onehot encoder in Spark, Sklearn based data preprocessing in AWS and ETL operations, feature extraction, PCA, built-in text to word embeddings trnaformation in Databricks are some notable examples. Here, datasets are consumed to produce new datasets with changed features.
  4. **Feature Extraction**: These activities create new features from the existing features of the dataset. Although there might be some correlation between the data attributes, these are useful for simpler models. Notable examples include feature store in Databricks and manual feature extractor in AWS. Old datasets are consumed to create new datasets with added features.
- **Model Training**: These transformations are employed to train models defined using a predefined architecture. Generally, the input is a dataset artifact and output is a trained model artifact. There still can be two further distinctions made:
  1. **Pure Model Training**: In this activity, only the simple transformation of using a dataset to train a model is performed. Model training step in Azure, Training using built-in algorithms, user input and estimators from other libraries in AWS, Spark Estimators, Trainer in GCP and Manual training in Databricks are some examples. Input artifacts are datasets, evaluation metrics and output artifacts are trained model, weights and hyperparameters.
  2. **Automatic Training**: These transformation are a cross between model training, transformation and assesment activities. The best example for these are AutoML [22] functionalities in Databricks, tuning using Hyperparameter Optimization (HPO) in AWS and tuner functionality in GCP. These transformations combine the training, testing and validation activities together, removing the need for connecting these three activities.
- **Model Inference**: Model inference operations, usually called Transform operations in AWS and GCP are used to batch transform a whole dataset using the trained model. This method is useful when the MLOps pipeline is part of a decision workflow and not just an endpoint being called by another service. The input is a dataset and a trained model and the output is a transformed dataset.

3.3.2 Assessment

Assessment operations take in artifacts such as machine learning models and create reports of their performance using different specified metrics. Model validation in Azure, ClarifyCheck and QualityCheck in AWS, which compare the

model with the metrics of the previous iterations of the training process, Evaluator in GCP and Scoring pipeline in Databricks.

3.3.3 *Decision*

Decision activities use the policies to decide between different pipeline branches and create the relevant artifact. for example, it can choose the branch of deployment if the machine learning model has higher then baseline accuracy. Condition and Fail steps in AWS are explicitly defined Decision activities. Most of the time decision activities and assessment activities are combined together to select the best model from different training runs. Decision models have the input of reports and metrics and their output are branch decisions.

3.3.4 *Provision*

Provision operations read the artifacts such as templates, workspace specifications and pipeline configurations to create the underlying infrastructure for the whole MLOps process. Deployment of models, which is usually done at the end stages of pipeline, also falls under provision. Provision activities mainly contain 3 types of activities:

1. **Model Registration**: Model registration activities are triggered by decision activities after all the correct criterion for the model metrics are passed by a newly trained model. The model is "registered" into the framework registry for storage and deployment.

2. **Model Deployment**: The registered model is deployed to an endpoint. The specifics of this have to be defined by user. This allows the model to be used via an endpoint.

We use these basic activities to define the different stages in any generic MLOps pipeline. But before that we must define the Core domain model of the MLOps pipeline.

3.4 Core Domain

The aim of the core domain model is to help in understanding the structure and hierarchy of the generalized pipeline. We use the methodology from [17] to construct the core domain model.

- **Pipeline**: A pipeline represents the whole MLOps pipeline.

- **Stage**: One or more stages make a pipeline. Each stage has different functions.

- **Activity**: Activities are the basic building block of any stage. They can be any of the four specializations. To represent the flow between the activities, there are two flow dependency types.

- **Preconditions**: Every activity has some preconditions which can be satisfied by artifacts, reports or infrastructure provisions.

- **Logical Dependency**: They represent the control flow in which the activities should take place because of their dependencies on each other or as a result of decision activities.

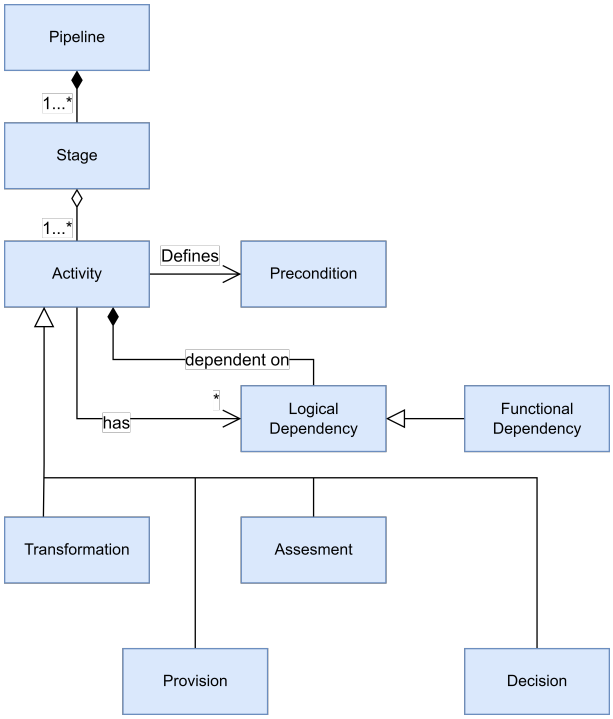

Figure 8: Pipeline Structure Core Domain

- **Functional Dependencies**: They represent the data flow between the activities. If the data from one activity goes to another activity, then there is a functional dependency between the activities. Functional dependency also implies logical dependency.

3.5 Generalized Stages

From the pipeline stages of MLOps providers in Section 2 and interpolating the pipeline and pipeline flows with the core domain model in 3, we can define a general pipeline structure with stages for MLOps. These stages can be ordered and are dependent on each other.

- **Provisioning**: This stage consists of "provision" activities. The design can be in such a way that all the infrastructure required can be provisioned before the pipeline starts or it can be provisioned just-in-time. So not all the Cloud providers may have a provision stage as the first stage, it can run in parallel as well.

- **Data Processing**: Transformation activities such as Data Retrieval, Data Analysis, Data Preprocessing and Feature extractions which convert the data from one form into another are included in Data Processing stage. This stage consumes data sources and creates a ready dataset for the stage.

- **Model Build and Training**: Model construction, and model training on the datasets. This is one of the core parts of any MLOps pipeline. This stage includes the Pure Model training activity. It consumes the datasets and metric policies from the user and produces a trained model.

- **Model Tuning and Validation**: In this stage the models metrics are calculated using different validation meth-

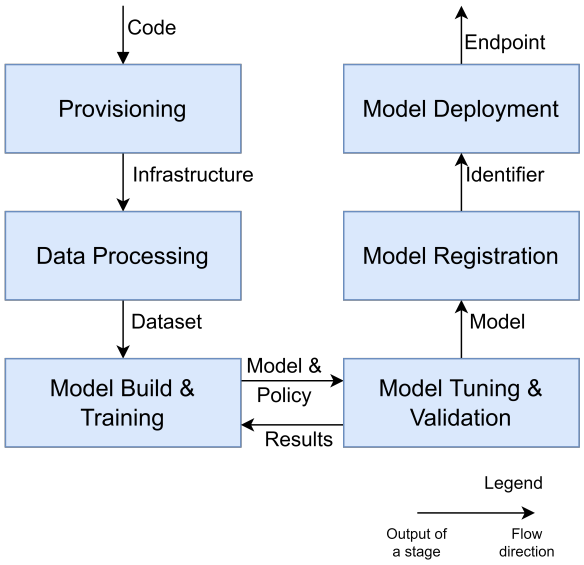

Figure 9: Generalized MLOps stages

ods. This is also the stage where most of the "decision" activities take place, since it involves comparing the model against the baselines, thresholds and the previous iterations of the model. This consumes trained models from the Model Build Training set and produces their reports using Assessment activity. Automatic Training activities combine the Model Build Training and Model Tuning Validation stages into one stage.

- **Model Registration**: Every "Successful" version of the trained model is registered in a central registry or repository. This is to keep track of the models and their versions. This stage consumes a validated model, stores the model data in a registry and produces a registry identifier.

- **Model Deployment**: Model deployment involves setting up the endpoints for accessing the model via some interface (commonly HTTP interfaces) and deploying the model and server application on a server. It consumes model and a registry identifier and produces an infrastructure endpoint.

4. CONCLUSION

In this paper, we discussed MLOps terminology, reviewed five workflows from MLOPs providers, and summarized the examined workflows into one general pipeline structure. Furthermore, we investigated the infrastructure of the different MLOps pipelines starting from data ingesting to model deployment. We then generalize the studied models into one model with four activities while defining a generic pipeline with six stages.

In the future, work can be focused on optimizing the interfaces between the activities and coupling of the stages.

5. REFERENCES

[1] Apache Spark™ - Unified Engine for large-scale data analytics. https://spark.apache.org/.

[2] Data Lakehouse Architecture and AI Company. https://databricks.com/.

[3] Databricks Feature Store | Databricks on AWS. https://docs.databricks.com/applications/machine-learning/feature-store/index.html.

[4] Databricks Machine Learning. https://databricks.com/product/machine-learning.

[5] MLOps: Continuous delivery and automation pipelines in machine learning. https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning.

[6] What Is Amazon SageMaker? - Amazon SageMaker.

[7] F. Almeida and G. Xexéo. Word embeddings: A survey. *arXiv preprint arXiv:1901.09069*, 2019.

[8] balapv. How Azure Machine Learning works (v2) - Azure Machine Learning. https://docs.microsoft.com/en-us/azure/machine-learning/concept-azure-machine-learning-v2.

[9] J. Bergstra, D. Yamins, and D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, pages 115–123. PMLR, 2013.

[10] Blackmist. Tutorial: AutoML- train no-code classification models - Azure Machine Learning. https://docs.microsoft.com/en-us/azure/machine-learning/tutorial-first-experiment-automated-ml.

[11] Blackmist. What is automated ML? AutoML - Azure Machine Learning. https://docs.microsoft.com/en-us/azure/machine-learning/concept-automated-ml.

[12] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching Word Vectors with Subword Information. *arXiv:1607.04606 [cs]*, June 2017. arXiv: 1607.04606.

[13] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[14] T. Chen and C. Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. Association for Computing Machinery. event-place: San Francisco, California, USA.

[15] T. Chen and C. Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[16] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[17] J. Döring and A. Steffens. An Architecture for Self-organizing Continuous Delivery Pipelines. *SWC*. https://www.swc.rwth-aachen.de/thesis/architecture-self-organizing-continuous-delivery-pipelines/.

[18] Ed Price. Machine Learning operations maturity model - Azure Architecture Center. https://docs.microsoft.com/en-us/azure/architecture/example-scenario/mlops/mlops-maturity-model.

[19] K. P. F.R.S. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[20] C. A. Gomez-Uribe and N. Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)*, 6(4):1–19, 2015.

[21] J. A. Hanley and B. J. McNeil. The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology*, 143(1):29–36, 1982.

[22] X. He, K. Zhao, and X. Chu. AutoML: A Survey of the State-of-the-Art. *Knowledge-Based Systems*, 212:106622, Jan. 2021. arXiv: 1908.00709.

[23] D. Kreuzberger, N. Kühl, and S. Hirschl. Machine learning operations (mlops): Overview, definition, and architecture. *arXiv preprint arXiv:2205.02302*, 2022.

[24] lgayhardt. What are machine learning pipelines? - Azure Machine Learning. https://docs.microsoft.com/en-us/azure/machine-learning/concept-ml-pipelines.

[25] N. Mehrabi, F. Morstatter, N. Saxena, K. Lerman, and A. Galstyan. A Survey on Bias and Fairness in Machine Learning. *arXiv:1908.09635 [cs]*, Jan. 2022. arXiv: 1908.09635.

[26] A. N. Modi, C. Y. Koo, C. Y. Foo, C. Mewald, D. M. Baylor, E. Breck, H.-T. Cheng, J. Wilkiewicz, L. Koc, L. Lew, M. A. Zinkevich, M. Wicke, M. Ispir, N. Polyzotis, N. Fiedel, S. E. Haykal, S. Whang, S. Roy, S. Ramesh, V. Jain, X. Zhang, and Z. Haque. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *KDD 2017*, 2017.

[27] NilsPohlmann. Create and run ML pipelines - Azure Machine Learning. https://docs.microsoft.com/en-us/azure/machine-learning/how-to-create-machine-learning-pipelines.

[28] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[29] V. Perrone, H. Shen, A. Zolic, I. Shcherbatyi, A. Ahmed, T. Bansal, M. Donini, F. Winkelmolen, R. Jenatton, J. B. Faddoul, B. Pogorzelska, M. Miladinovic, K. Kenthapadi, M. Seeger, and C. Archambeau. Amazon SageMaker Automatic Model Tuning: Scalable Gradient-Free Optimization. *arXiv:2012.08489 [cs, stat]*, June 2021. arXiv: 2012.08489.

[30] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You Only Look Once: Unified, Real-Time Object Detection. *arXiv:1506.02640 [cs]*, May 2016. arXiv: 1506.02640.

[31] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, oct 2016.

Towards an overview of metrics in recent years for microservice architectures

Savas Köklü
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
savas.koeklue@rwth-aachen.de

ABSTRACT

Microservices are becoming an increasingly preferred software architecture pattern style for companies and developers. Although it offers many solutions to different problems in the field of software development, this architectural pattern is missing a clear definition. There are still many uncertainties in measuring the quality of microservice-based programs and their architecture. Evaluating the quality of an architecture is particularly important during design and implementation in order to be able to detect errors early on and intervene before the MSA can be deployed. For better assessment of the quality of a microservice architecture, models and metrics have been established, discussed and many advances have been made in this area, especially in the research of the last years. In this paper we are reaching towards an overview of the research of recent years and how it focuses on quality measurement using design documents or source code of a microservice architecture. For this purpose, we have extracted 16 metrics. For each metric, we give a description, relate it to quality attributes and match it to the design documents or source code of an architecture. We present these characteristics in two tables as an overview for researchers and microservice practitioners.

Keywords

microservice architecture, microservice, metrics for microservice architecture, quality measurement microservice-based applications

1. INTRODUCTION

In 2020, a report published by O'Reilly found that microservice architecture is becoming increasingly popular in the software industry, with about 77% of companies using microservice architectures [23]. Although it gained a lot of interest, there is still no clear definition for this architecture in the literature. However, in this paper I follow the definition provided by Dragoni et al. [13] that a microservice architecture (MSA) is a collection of small services with specific tasks that interact through shared communication channels to achieve business goals. According to the authors of [13] the main advantages of MSAs are services with high granularity, support for different technologies, easy swapping between services, a high fault tolerance and decentralized information.

If an architecture is not properly designed, many disadvantages can arise. Kalske et al. argue that the MSA could become too complex than necessary or its maintenance could take longer than appropriate [19]. As an example, the authors state that instead of using similar or same technologies, different ones could be used, so that changes to individual services and thus also to the MSA could take longer. The advantages mentioned above might not be used to their full extent or even be reversed. Creating a system and its architecture is a very difficult and important task, especially if the project can become very large [12]. MSAs can consist of hundreds or thousands of services [10]. Before the system can be developed, the architecture must be carefully designed and planned. During implementation, attention must be paid to ensure that the architecture is not changed in the wrong way [12].

Bogner et al. identified in 2017 in [9] that there was a lack of research on measuring the quality of MSAs at that time. With this work, I want to investigate in which direction the research of the last years on metrics of MSAs is developing. For this purpose, I do a lightweight systematic literature review to answer the following research question RQ1:

- What metrics are present in the research of recent years for the design documents or source code of a MSA and for which quality attributes are these metrics used?

As a result, I give an overview of the found metrics.

The rest of the work is structured as follows. Section two begins by setting out the framework for the rest of the paper and explaining basic concepts and terms. Then, in section three, I look at research that is close to mine. In section four, I describe my approach and the design of the study. The result and discussion comes in section five. Then I close the study with a conclusion and ideas for future work.

SWC Seminar 2022 RWTH Aachen University, Germany.

2. BACKGROUND

2.1 Software quality and quality attributes

The ISO standard 9000 defines quality as the "degree to which a set of inherent characteristics of an object fulfils requirements". Objects can be goods, raw materials, but also services, work processes and the content of designs and concepts [16]. To measure the fulfilment of requirements and thus the quality of an MSA, I measure the quality attributes of it. To define the quality attributes, I refer to the ISO/IEC standard 25010 [17].

The ISO/IEC standard 25010 [17] defines a model that specifies the quality attributes of a software product. It divides software product quality into the quality attributes functional suitability, performance efficiency, compatibility, usability, reliability, security and maintainability. Each of these quality attributes is broken down into smaller quality factors. Because these quality factors are granular and specific, they enable measurement of the quality attributes and thus the quality of a software product. For example, maintainability as a quality attribute is defined in the standard as "the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements". Maintainability is subdivided into the quality factors modularity, reusability, analysability, modifiability and testability.

To answer the research question RQ1 and thus get a better overview of the found metrics, I assign them to the quality attributes from the ISO/IEC standard 25010 [17].

2.2 Software architecture

By focusing on a system's "big picture" in a software architecture, software architects can abstract away the finer details of implementation and other development parts [25]. The system's software architecture can be defined as follows: "The structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them" [7]. An architectural description can include multiple views that emphasize different aspects of a system, such as logical, implementation, deployment, or process views, as well as perspectives from different stakeholders, such as developers, end-users, business analysts and project managers. For example, The implementation view models the source code of a software architecture and typically shows technical components with their properties and relationships to each other [21]. In this work, I focus on the architecture information described in higher-level structures with reference to the implementation view. Thus, I consider the software architecture of a microservice-based application as a *component and connectors graph*. I see microservices as *components* that represent system functionality, units of computation or data storage, and *connectors* link them together. Such a graph abstracts the to-be-implemented source code of an MSA. A *component* consists of more technical components or classes and is characterised by the fact that it can be deployed independently and thus has no in-memory connection to any other *component* of the MSA. The *components* offer request possibilities to the other *components* of the MSA [33].

2.3 Metrics

A key need in all engineering fields is right measurement, and for software engineering it is the same [15]. According to the IEEE glossary, a metric is a "quantitative measure of the degree to which a system, component, or process possesses a given attribute". In the definition it refers to quality metrics which are defined as a "quantitative measure of the degree to which an item possesses a given quality attribute". A quality metric is a "function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given quality attribute". Thus, in the terms of the IEEE definition of a quality metric, a software engineering metric is the mapping of software or a process of software development to a vector or scalar quantity [1]. Often, a metric includes an understanding of his representation [21].

In the context of my work, I use the source code or design documents of a software architecture of a microservice-based application as input for a metric and get a scalar or vector size for it. I see the source code of a software architecture as the concrete implementation of the application. To define design documents, I refer to the IEEE standard 1471 definition of the design of an architecture. The design of an architecture is described as a "collection of products for documenting an architecture" [2]. The design documents could, for example, have been created using an architecture description language such as UML [21].

3. RELATED WORK

There are many studies related to the quality measurement of software architectures, such as by Coulin et al. [12] and by Stevanetic and Zdun [30]. In [12] Coulin et al. collect metrics to measure the quality of the design of a software architecture and in [30] Stevanetic and Zdun consider also the implementation. Five quality attributes were found by Coulin et al., maintainability, extensibility, simplicity, reusability and performance, and a wide range of metrics to measure them. I do not consider software architectures in general, but restrict myself to microservice architectures. Also, the paper [12] were published in 2019 and [30] in 2015, so I review newer work.

Research papers that use metrics to evaluate microservice architectures are for example [4] by the authors Al-Debagy and Martinek or [27] written by Pautasso and Wilde. Al-Debagy and Martinek collect in [4] metrics for evaluating the design of microservice architectures. They focus on cohesion, granularity and complexity and found some similar metrics like the number of operations or lack of cohesion metric (LCOM) as I did. However, these studies mostly focus on one quality attribute, such as maintainability, and do not attempt to provide an overview.

Another area when it comes to microservice architectures is the consideration of patterns. There is a lot of research around collecting and analysing patterns, as Ntentos et al. in [24] describe. In the work [24] metrics for assessing architecture conformance to microservice architecture patterns and practices are collected. In this paper, I do not look at specific patterns and, how they are used in an architecture, but I give an overview of general metrics for measuring quality.

Furthermore, there are papers for example by Cojocaru et al. [11] and [33] written by Zdun et al. that focus on the eval-

Table 1: Overview of the Extracted Metrics

Metrics Name	Measurement Object	Quality Attributes	Reviewed	Cited
Lines of Code	Source code	maintainability	[11]	[9]
Lines of Code Average	Source code	maintainability	[11]	
Number of Open Interfaces	Source code, Design*	maintainability	[3],[11]	[14]
Number of Request Option Usages	Source code, Design*	maintainability	[11]	[14]
Ratio of Dependencies	Source code	maintainability	[11]	[8]
Number of Technologies Used	Source code, Design*	maintainability	[11]	[14]
Number of Asynchronous Dependencies	Source code, Design*	maintainability	[11]	[14]
Resiliency to Failure	Source code	reliability*	[11]	[5]
Request Options Multiplied Methods	Source code, Design*	maintainability	[11]	
Stateless	Source code, Design*	maintainability, reliability*	[11]	
Number of Stateless	Source code, Design*	maintainability, reliability*	[11]	
GitHub Stars	Source code	maintainability	[11]	
Number of Parameters	Source code, Design*	maintainability	[3]	
Parameters Used	Source code, Design*	maintainability	[3]	
Number of Parameters Used Maximum	Source code, Design*	maintainability	[3]	
Lack of Cohesion Metric (LCOM)	Source code, Design*	maintainability	[3]	

uation of microservice architectures after the decomposition from large tightly coupled designed applications. Compared to Zdun et al., Cojocaru et al. present the smallest subset of the metrics available in the literature to evaluate microservice architectures generally in the industry. Most of these focus on maintainability. I have also adopted many of the metrics found by Cojocaru et al.. However, as the study is from 2019, I have also found other metrics.

4. STUDY DESIGN

To answer the research question RQ1, this study was conducted with the guidelines of [20]. I have made an effort to make it reproducible and transparent.

4.1 Search and selection process

I used IEEE as a search platform for my research. It is one of the most important computer science research databases. My query was the following:

```
("All Metadata":"microservice* architecture*" OR
"All Metadata":"microservice* application*")
AND ("All Metadata":metric*)
AND ("All Metadata":evaluation OR "All Metadata":quality OR "All Metadata":assessment)
AND ("Index Terms":"microservice* architecture*" OR
"Index OR "Index Terms": "microservice application" OR "Index Terms": "microservice applications" OR "Index Terms": "microservices application" OR
"Index Terms": "microservices applications")
AND ("Index Terms":metric OR "Index Terms":metrics)
```

The search found twelve papers. Due to limited scope of this work, I included papers that met the following criteria:

- the publication date is after 2018
- the paper contains metrics about the source code or design documents of a MSA

After applying the two selection criteria, I had selected two papers for further processing. Based on the first criterion, one paper from 2017 and one from 2018 were not included and based on the second criterion, I excluded eight other papers. Due to the limited extent of this work, I did not make any further attempts to select other papers, for example, I did not examine the citations of the papers I found. For the same reason, I did not perform a quality assessment of the two selected papers.

4.2 Data extraction

The two papers that resulted from the selection process contained 16 different metrics that can be applied to the source code or design documents of a MSA. I summarised the extracted information about these metrics in the tables 1 and 2. Table 1 gives an overview of the metrics found and their properties, and table 2 gives a description of the metrics. The columns of the two tables are the following:
Metrics Name: The name of the metric presented in the row of the table. If the paper presenting the metric has not named it, then I have meaningfully given it a name considering the extracted information.
Measurement Object: Indicates whether the metric can be applied to the source code or the design documents of an architecture. Because of size constraints, I have only used the word design in the table instead of design documents. In addition to the application mentioned in the papers, I have also added other possible applications of the metric. These added applications are based on my own experience and are marked with the symbol *.
Quality Attributes: For each metric, I have taken the quality attributes from the papers and added some of my own consideration. The additional added quality attributes are marked with *.
Reviewed: The papers resulted from the selection process that describe the metric are listed here.
Cited: If the metric from the two selected papers was cited from another source, I have indicated those sources here.
Description: Extracted information from the two papers found.

5. RESULTS AND DISCUSSION

As a result, to answer the research question RQ1, I build table 1 and 2 as an overview of MSA metrics that have been discussed in the literature in recent years. I selected

Table 2: Metrics Description

Metrics Name	Description
Lines of Code	The lines of code are incremented to measure the size of the architecture or a microservice.
Lines of Code Average	The lines of code of the microservices in the microservice architecture are incremented and then averaged. This metric can be used, for example, to find large microservices in the microservice architecture and possibly split them into several.
Number of Open Interfaces	The request possibilities provided by the microservices in a microservice architecture are listed. These can be for use in their microservice architecture as well as for other applications. The level of detail of the request, i.e. whether to count parameters that are passed in a request, can be decided at one's discretion. If the number of operations is only considered and it exceeds 10, the microservice should be split.
Number of Request Option Usages	The total frequency with which the provided request options from a microservice are used by other microservices.
Ratio of Dependencies	The number of request options provided by a microservice is divided by the total frequency with which these request options are used by other microservices in the code. Thus, it shows the ratio of the two quantities and calculates the degree of coupling for a microservice.
Number of Technologies Used	The various technologies used to implement a microservice are counted up. By technologies is meant which languages and frameworks are used, but other technologies can also be included at your discretion. A high count negatively influences maintainability.
Number of Asynchronous Dependencies	The number of request options a microservice uses while other code of it can be executed.
Resiliency to Failure	The metric can only take the value True or False. If an error occurs, the microservice is able to restart if necessary and call up the last state before the error.
Request Options Multiplied Methods	The number of request options provided by a microservice is multiplied by the number of methods needed to implement it. The higher the result, the more granular the microservice is. The degree of granularity must not be too high but not too low in order to positively influence maintainability.
Stateless	It is specified whether a microservice is stateless. A stateless application does not change its behaviour with incoming requests. Such Stateless applications can then be enlarged better, e.g. by duplicating the microservice.
Number of Stateless	In the table, the metric Stateless is listed. This is the number of microservices for which Stateless is true. Microservice architectures that have many stateless microservices are more scalable and thus more maintainable.
GitHub Stars	The code of microservices or the entire architecture is published publicly on GitHub. Users can then reuse and rate the code there. This rating of the individual users is summarized by GitHub into a star rating and thus indicates the reusability.
Number of Parameters	All unique parameters that can be used for a request option provided by a microservice are counted. The larger the value, the potentially larger the microservice and thus more potential negative impact on maintainability.
Parameters Used	For each request possibility, the supported parameters are summed up. The larger the value at the end, the potentially larger the microservice and thus more potential negative impact on maintainability.
Number of Parameters Used Maximum	For a microservice, the number of request possibilities is multiplied by the previously defined Number of Parameters metric. The result represents the maximum that the Parameters Used metric can have, which is the case if all request possibilities provide the same parameters.
Lack of Cohesion Metric (LCOM)	The metric indicates how the operations or request options in a microservice relate to each other in terms of functionality. The metrics, Parameters Used and Number of Parameters Used Maximum, are given in the table. Parameters Used is divided by Number of Parameters Used Maximum and then 1 minus the result is calculated. The final result should be between 0 and 0.8 otherwise it means a bad cohesion for the microservice and it should be split. A bad cohesion also affects the maintenance badly.

two relevant papers out of the ten. In total, I found 16 metrics of which 13 metrics are just for the measurement of maintainability, one just for the measurement of reliability and two just for the measurement of maintainability and reliability. All metrics I found were applicable to the source code as referenced in the two papers. For ten of the 16 metrics, I identified that they could potentially be applied to given design documents.

Eight out of the ten papers I found with my query were about analysing an MSA in deployment and only two focused on the source code of an MSA. This coincides with the findings of other authors in papers published before 2019 like Alshuqayran et al. in [6] or Pietrantuono et al. in [28]. They have come to the same conclusion as I have, that the focus of the research is on the analysis of an MSA at runtime. Alshuqayran et al. [6] conducted a systematic mapping study of microservice architectures in 2016. Their research focused on the years 2014 to 2016. Before 2014, the term MSA was not widely used in the literature [26]. They discovered that in 33 papers found, 31 discussed the topics of "deployment", "cloud" and "performance". As described in 2018 by Pietrantuono et al. [28], many papers look at runtime properties of MSAs. Using tools such as AWS Cloud Watch, it is very popular to monitor performance between service invocations and runtime errors [29]. Too few papers are published that present metrics on the source code of an MSA, and in particular there was no metric described in the found papers for analysis of design documents.

The metrics found were almost all about the measurement of maintainability and there were hardly any for the other quality attributes from the ISO/IEC standard 25010 [17]. These findings coincide with the findings of Coulin et al. [12] for software architecture metrics in general. Thus, most of the metrics they found focused on the maintainability, in particular the coherence and coupling, of a software architecture. Coulin et al. [12] reasoned that maintainability is the most important quality attribute of a software architecture and that all others depend on it. When looking at MSAs, Valdivia et al. also states that maintainability is mainly measured in the existing literature [31]. I can only agree with Coulin et al. [12] that maintainability is the most important quality attribute of a software architecture and thus also of an MSA. For example, maintainability measures the granularity of an MSA with the quality factor modularity. If maintainability is very high, the structure of the MSA itself, i.e. how the microservices interact individually and with the others, is also well constructed [18]. Furthermore, maintainability is often a synonym for sustainability, as it allows software programs to be updated and used for many years [22]. However, I think that this is not a reason why the other quality attributes have hardly been researched in recent years. The research needs to be more extensive, because it is also important to be able to measure the quality of an MSA through the other quality attributes. For example, it is relevant to consider performance efficiency before deploying the MSA to prevent poor performance early on. If it later becomes apparent that a poor architecture is the reason for insufficient performance, then this can only be repaired with great difficulty [32].

6. CONCLUSION AND FUTURE WORK

In software development, architecture is a key factor [12]. With this paper, I have conducted a lightweight systematic literature review towards finding out which metrics for measuring which quality attributes of a microservice architecture have been discussed in recent years. Out of ten papers I found with my query, only two contained metrics for the source code or design documents of a microservice architecture. From these two papers I extracted 16 metrics. The metrics were mainly focused on measuring the maintainability of a microservice architecture and I hardly found any for the other quality attributes. Furthermore, the metrics from the papers were only mentioned for the application on the source code of a microservice architecture. My result can be used as a starting point to get an overview of MSA metrics present in the literature of recent years.

Research in the future can focus on doing a more comprehensive systematic literature review, so that the search can be extended to other platforms such as ACM. In addition, papers could also be reviewed before 2019 to build a complete catalogue of metrics. Finally, it is necessary to find more metrics for source code or especially design documents, so that quality attributes other than maintainability of a microservice architecture can be measured.

7. REFERENCES

[1] IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[2] IEEE recommended practice for architectural description for software-intensive systems. *IEEE Std 1471-2000*, 2000.

[3] O. Al-Debagy and P. Martinek. Extracting microservices' candidates from monolithic applications: Interface analysis and evaluation metrics approach. In *2020 IEEE 15th International Conference of System of Systems Engineering (SoSE)*, pages 289–294, 2020.

[4] O. Al-Debagy and P. Martinek. A metrics framework for evaluating microservices architecture designs. *Journal of Web Engineering*, pages 341–370, 2020.

[5] N. Alshuqayran, N. Ali, and R. Evans. A systematic mapping study in microservice architecture. In *2016 IEEE 9th international conference on service-oriented computing and applications (SOCA)*, pages 44–51. IEEE, 2016.

[6] N. Alshuqayran, N. Ali, and R. Evans. A systematic mapping study in microservice architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51, 2016.

[7] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[8] J. Bogner, S. Wagner, and A. Zimmermann. Automatically measuring the maintainability of service- and microservice-based systems: A literature review. In *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, IWSM Mensura '17, page 107–115, New York, NY, USA, 2017. Association for Computing Machinery.

[9] J. Bogner, S. Wagner, and A. Zimmermann. Automatically measuring the maintainability of service-and microservice-based systems: a literature review. In *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, pages 107–115, 2017.

[10] J. Bogner, S. Wagner, and A. Zimmermann. Towards a practical maintainability quality model for service-and microservice-based systems. In *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*, pages 195–198, 2017.

[11] M.-D. Cojocaru, A. Uta, and A.-M. Oprescu. Attributes assessing the quality of microservices automatically decomposed from monolithic applications. In *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 84–93, 2019.

[12] T. Coulin, M. Detante, W. Mouchère, and F. Petrillo. Software architecture metrics: a literature review. *arXiv preprint arXiv:1901.09050*, 2019.

[13] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham, 2017.

[14] T. Engel, M. Langermeier, B. Bauer, and A. Hofmann. Evaluation of microservice architectures: a metric and tool-based approach. In *International Conference on Advanced Information Systems Engineering*, pages 74–89. Springer, 2018.

[15] N. Hina, H. Babur, H. AbuBakar, W. Tamoor, and R. Nasim. Software metrics: Investigating success factors, challenges, solutions and new research directions. *INTERNATIONAL JOURNAL OF SCIENTIFIC TECHNOLOGY RESEARCH VOLUME 9*, 2020.

[16] ISO 9000. Standard, International Organization for Standardization, Geneva, CH, 2015.

[17] ISO/IEC 25010. Standard, International Organization for Standardization, Geneva, CH, 2011.

[18] ISO/IEC 25010. ISO/IEC 25010:2011, systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models, 2011.

[19] M. Kalske, N. Mäkitalo, and T. Mikkonen. Challenges when moving from monolith to microservice architecture. In *International Conference on Web Engineering*, pages 32–47. Springer, 2017.

[20] S. Keele et al. Guidelines for performing systematic literature reviews in software engineering. Technical report, Technical report, Ver. 2.3 EBSE Technical Report. EBSE, 2007.

[21] J. Ludewig and H. Lichter. *Software Engineering – Grundlagen, Menschen, Prozesse, Techniken / Jochen Ludewig ; Horst Lichter*. 01 2013.

[22] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng. Decoupling level: a new metric for architectural maintenance complexity. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 499–510. IEEE, 2016.

[23] A. Muaz, M. E. Rana, and V. A. Hameed. A framework for catering software complexity issues using architectural patterns. In *2021 Third International Sustainability and Resilience Conference: Climate Change*, pages 554–561, 2021.

[24] E. Ntentos, U. Zdun, K. Plakidas, S. Meixner, and S. Geiger. Metrics for assessing architecture conformance to microservice architecture patterns and practices. In *International Conference on Service-Oriented Computing*, pages 580–596. Springer, 2020.

[25] P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and their Applications*, 14(3):54–62, 1999.

[26] C. Pahl and P. Jamshidi. Microservices: a systematic mapping study. *CLOSER (1)*, pages 137–146, 2016.

[27] C. Pautasso and E. Wilde. Why is the web loosely coupled? a multi-faceted metric for service design. pages 911–920, 01 2009.

[28] R. Pietrantuono, S. Russo, and A. Guerriero. Run-time reliability estimation of microservice architectures. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 25–35, 2018.

[29] C. Richardson. Application metrics. `http://microservices.io/patterns/observability/application-metrics.html`. Accessed: 2022-06-07.

[30] S. Stevanetic and U. Zdun. Software metrics for measuring the understandability of architectural structures: A systematic mapping study. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, EASE '15, New York, NY, USA, 2015. Association for Computing Machinery.

[31] J. A. Valdivia, X. Limón, and K. Cortes-Verdin. Quality attributes in patterns related to microservice architecture: a systematic literature review. In *2019 7th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pages 181–190, 2019.

[32] L. G. Williams and C. U. Smith. Performance evaluation of software architectures. In *Proceedings of the 1st International Workshop on Software and Performance*, WOSP '98, page 164–177, New York, NY, USA, 1998. Association for Computing Machinery.

[33] U. Zdun, E. Navarro, and F. Leymann. Ensuring and assessing architecture conformance to microservice decomposition patterns. In *International Conference on Service-Oriented Computing*, pages 411–429. Springer, 2017.

Modelling portability and maintainability in Microservice-based applications

Merzough Badry Münker
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
badry.muenker@rwth-aachen.de

Jurgen Abazi
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
jurgen.abazi@rwth-aachen.de

ABSTRACT

The development of Microservice applications brings new challenges for quality assurance. Especially as Microservice architecture (MSA) continues to spread in professional software development, it is necessary to ensure MSA's long-term maintainability and portability. Currently, there is little research on the influence of MSA on quality attributes and on how maintainability and portability can be measured and evaluated in MSA. This paper surveys the literature's proposed metrics for maintainability and portability. A mapping for Quality-Models from service-oriented architecture (SOA) and service-Based Architecture (SBS) to MSA is proposed. In addition, a visual representation is presented to illustrate the influence of quality factors on quality attributes. As a result, it can be stated that this field has not yet been sufficiently explored.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.8 [**Software Engineering**]: Metrics—*maintainability, portability*

Keywords

Microservices, Maintainability, Portability, SLR

1. INTRODUCTION

In recent years, Microservice Architecture has gained popularity as a software architecture pattern and the adoption and success has grown rapidly [9] [20]. Martin Fowler and James Lewis have defined the Microservice architectural style "as an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API" [9]. Additionally, the terms Portability and Maintainability refer to the quality attributes defined in ISO/IEC 25010 standard, which defines a high-level quality model for software products. The quality characteristics defined by the model are the basis of the research, which aims to study how they can be defined and modeled in Microservice Architecture. The ISO/IEC 25010 standard [13] defines portability and maintainability as followed:

- **Portability** as "the degree of effectiveness and efficiency with which a system, product, or component can be transferred from one hardware, software, or other operational or usage environment to another."
- **Maintainability** as "the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in the environment and the requirements."

The quality model defined in the ISO/IEC 25010 standard determines which quality characteristics will be considered when evaluating a software product's properties. The model is used as the basis for defining portability and maintainability, as it is widely accepted by both industry experts and academic researchers [7].

The paper aims to analyze and model maintainability and portability in the context of Microservice-based applications. The Systematic Literature Review (SLR) methodology was applied to identify and synthesize research regarding the topic to achieve the goal. Based on the extracted data, an overview of the measurement of portability and maintainability in the context of the MSA was provided. This paper's primary contribution is to aggregate the existing metrics for maintainability and portability and propose a model to measure the quality attributes.

Section 2 provides background information necessary to better understand the context of the paper. Section 3 outlines studies related to the topic of this paper. Section 4 defines the research questions and explains in detail the research method used in the study. Section 5 presents the findings and the results of the study and answers the specified research questions. Section 6 discusses the results in the context of existing research. Section 7 gives a conclusion and outlook on further research.

2. BACKGROUND

Microservice architecture (MSA), service-oriented architecture (SOA) and service-based architecture (SBA) are often overloaded, with each term having multiple definitions and variations [22]. In the following subsections, we present general definitions.

SWC Seminar 2022 RWTH Aachen University, Germany.

2.1 SBS, SOA and MSA

SBS is a type of architectural pattern that, unlike monolithic systems, strongly emphasizes service as the primary architectural component of a system. Both MSA and SOA are often considered special types of service-based architectures. A common characteristic of these architectures is that they are distributed. [27]

SOA is a design approach where multiple services collaborate to provide some end set of capabilities. A service here typically means an entirely separate operating system process. The communication between these services is done by calls over a network and not by method calls within a process boundary. [27]

MSA was described by James Lewis and Martin Fowler as "an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and are independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies" [23].

2.2 Comparison

An SBS is a superordinate concept for SOA and MSA. Therefore, an SOA or MSA can be understood as an SBS specialization. Similar to the definition of SOA and MSA, the distinction between the concepts are unclear. [23]

One possible generalization is to regard MSA as a subform of SOA with additional restrictions. MSA neglect central components such as the enterprise service bus or the business process-centric service choreography. Nevertheless, complement the latest implementation findings such as continuous deployment (CD) and DevOps. From this, the following relationship between the terminologies can be derived 1

Figure 1: relation between SBS, SOA and MSA

With this background, it is possible to map selected metrics for SBS and SOA to MSA. When mapping metrics from SOA to MSA, all metrics related to exclusive SOA concepts such as the Enterprise Service Bus are omitted.

3. RELATED WORK

Shanshan Li et al. did a systematic literature review on 2020 regarding quality attributes of MSA [19]. However, maintainability and portability were not among the attributes studied by the paper.

The first purpose-built quality model for maintainability metrics for SBS was created by Bogner et al [4] [3] [6]. In [4] and [3], they summarized existing literary metrics for OO and SBS. In [3], a notation is introduced to represent the influence of a quality factor on a quality attribute. Bogner et al. [3] agrees with Cardarelli [6] that the area of quality model for maintainability metrics needs further research and validation.

H. Ghandorh did a systematic literature review, A. Noorwali, A. B. Nassif, L .F. Capretz and R. Eagleson in 2020 regarding software portability and the metrics proposed to measure it [10]. The paper offers preliminary results in analyzing portability in a broader scope but is not focused on MSA. It must be noted that research on portability of MSA is lacking.

4. RESEARCH METHOD

The Systematic Literature Review methodology was followed according to the guidelines defined by Kitchenham et al.[14]. This section explains the following steps in detail.

4.1 Research Question

To obtain precise and suitable models of portability and maintainability, we formulated 2 research questions:

RQ1 - *What research studies have been done on portability and maintainability in the context of Microservice, Service-Based, and Serviced-Oriented Architecture?* Answering this question provides an understanding of the current state of research regarding the two quality attributes of interest. This question is an important step that can lead to an answer to the second research question.

RQ2 - *How to model portability and maintainability in the context of the quality of Microservice Architecture?* This is the central research question of this paper. It aims to explore the results of the first research question and use them as the foundation for developing precise, concise, and correct models of the quality attributes of portability and maintainability.

4.2 Search and Selection Process

4.2.1 Initial Search

An automatic search was done on three of the biggest scientific databases in Software Engineering, namely ACM Digital Library, IEEE Xplore, and Scopus. The meta-search string used as the basis in the automatic search of the different databases is detailed on listing 1:

```
(MSA OR "microservice" OR "micro-service"
OR "microservice" OR SBA OR SBS OR
"service-based" OR "service-based" OR soa
OR "service-oriented" OR
"service-oriented") AND (portable* OR
maintainabi*)
```

Listing 1: The Meta Query String

The search string was designed to limit the search space to MSA, SOA, and SBS results and either portability or maintainability (or both). In the initial phase of the research, the query was limited to Microservices only. However, the search space was broadened due to low results, especially regarding portability. The string was then updated to match the specific syntax of each database that it was used on. Finally, no time-span restrictions on the publication date of the results were placed; therefore, all papers available at the time of running the search string were considered.

4.2.2 Impurity and Duplicate Removal

The automatic search returned results that were not research papers. Such results were manually removed. In addition, papers that shared the same title, publication date, and author were considered duplicates and manually removed during this phase.

4.2.3 Inclusion/Exclusion Criteria

Inclusion and exclusion criteria were defined and applied to select the appropriate studies. Such selection criteria are outlined as follows:

Inclusion Criteria:

- **I1** - Paper covers maintainability or portability in the context of Microservices, Service-Oriented or Service Based Architecture.
- **I2** - Paper offers metrics for measuring maintainability or portability.
- **I3** - Paper suggests approaches for modeling maintainability or portability.

Exclusion Criteria:

- **E1** - Paper is not written in English.
- **E2** - Paper is not available as full-text.
- **E3** - Paper only mentions Microservice, Service Oriented, or Service Based Architecture in keywords or as an example.
- **E4** - Maintainability and portability are mentioned but not further analyzed.
- **E5** - Paper focuses on the application of a newly introduced architecture or framework to improve maintainability and portability.

4.2.4 Combination and Snowballing

After the selection process, all the remaining studies were combined together. However, because of the small number of remaining results, a snowballing process was performed to increase the set of potential studies that are relevant to the topic. The studies were read in detail, and their references were explored to find potentially interesting papers. The final set of documents is shown in table 1 in the appendix of this paper. The results of the SLR give an outlook on the current state of research regarding portability and maintainability and additionally serve the purpose of answering the first research question.

4.2.5 Data Extraction

In the last phase of collecting scientific works, the definition of attributes was investigated. The focus was on how an attribute influences the maintainability or portability of MSA. In addition, different ways were determined around the attributes to specify. Finally, in the absence of research, an attempt was made to create a mapping from a higher-level application structure to MSA. For example, from SBSs or SOA.

5. RESULTS AND FINDINGS

This section presents the results and findings from the literature review conducted. In the first subsection, a formal description of the relationship between quality attributes and quality factors is presented. In the second subsection, a quality model of maintainability based on the reviewed studies is presented, while a quality model of portability is given in the third subsection. Since there are no metrics for MSA, the metrics for SBS and SOA are mapped to MSA. This mapping is possible because, as described in section 2, an MSA is a specialization of SOA or SBS.

5.1 Formal description

Bogner et al. [3] provide the first formal description of the relationship and influence between a quality factor and quality attribute. In the formulation, characteristics are defined as product factors of a combination of an entity and a quality factor that affects a quality attribute. In the context of their work, the quality attribute was maintainability. The proposed notation is as follows:

$$[Entity|FACTOR] \xrightarrow{+|-} [QualityAttribute]$$

With a concrete example from Bogner et al.:

$$[Function|COMPLEXITY] \xrightarrow{-} [Analyzability]$$

The given example is interpreted as: The quality factor complexity of a function has a negative effect on the quality attribute analyzability.

5.1.1 Visual representation

Based on Bogner's formal description, we propose a visual representation in the form of a tree:

- The root node represents the quality attribute.
- The direct children of the root node represent a quality factor of the quality attribute.
- The direct children of the quality factor node represent a metric of the quality factor.
- If the quality factor negatively affects the quality attribute, the node has a dashed line.
- If the quality factor influences the quality attribute positively, the node has a solid line.

The figure 2 shows an abstract example.

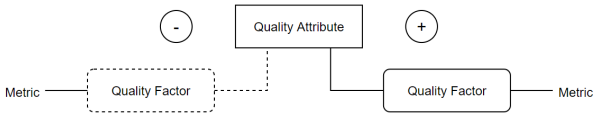

Figure 2: Abstract example for a quality model

5.2 Maintainability metrics for MSAs

Characteristics such as granularity, code maturation, size, complexity, coupling, and cohesion are frequently cited in literature dealing with maintainability. One of the first researchers to propose a quality model for maintainability of SBSs is Bogner et al. [6]. An overview and the relation of quality factors to the quality attribute maintainability is given in figure 3.

5.2.1 Granularity

Definition: Granularity is described as the size and the degree of decomposition of the services of a Service-Based System (and subsequently Microservice Architecture). Traditionally, the metric used to describe the granularity of systems is *Lines of Code* (LOC). However, because of the increased technological heterogeneity found in SBSs and MSAs, the metric is of little interest. Therefore a few other metrics are often suggested in studies more fitting to SBSs and MSAs, namely *Weighted Service Interface Count* and *Component Balance*.

$$[MSA|GRANULARITY] \xrightarrow{+} [Maintainability]$$

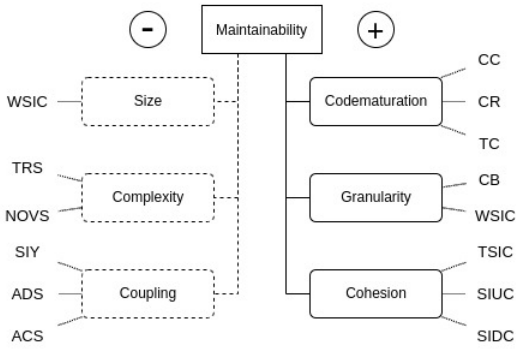

Figure 3: Quality Model for Maintainability

Weighted Service Interface Count (WSIC): WSIC(S) denotes the count of exposed interface operations of a service S. Additionally, the interface operations can be weighted in two ways: by their number or by the granularity of parameters with a default weight of 1.

Component Balance (CB): Component Balance defines the appropriateness of granularity at a system level. It can be defined as the product of two different measurements, specifically:

- *System Breakdown (SB)* which creates a value in the range $[0, 1]$ based on the count of top-level frameworks.
- *Component Size Uniformity (CSU)* which creates a value in the range $[0, 1]$ based on the Gini Coefficient of component volumes.

Studies often use Lines of Code to measure granularity which, as mentioned before, is not optimal due to microservice architectures being heterogeneous systems. Therefore, studies such as Bouwers in 2011 [5] suggest using Weighted Service Interface Count (WSIC). Additionally, CB must be initialized with an upper bound worst number of top-level components ($CB = 0$) and an optimal number of components ($CB = 1$).

5.2.2 Code Maturation

Definition: According to Bogner et al. [4], code maturity describes the degree of code quality in the areas of technical proficiency and consistency. A high code maturation improves the maintainability of an MSA.

$$[MSA|CODE\ MATURATION] \xrightarrow{+} [Maintainability]$$

Bogner et al. [4] has identified three metrics to determine the code maturation.

Comment Ratio (CR): Following Fluri et al. [8] findings, Bogner et al. [4] proposes the Comment Ration CR(S) metric. The metric is defined as the ratio between the number of comment lines and the total number of lines.

$$CR(MSA) = \frac{LOC_{comment}(MSA)}{LOC(MSA)}$$

Clone Coverage (CC): Following Koschke et al. [15] findings, Bogner et al. [4] proposes the Clone Coverage CC(S) metric. The metric is defined as the ratio between the duplicated lines and the total number of lines.

$$CC(MSA) = \frac{LOC_{duplicated}(MSA)}{LOC(MSA)}$$

The MSA is harder to maintain with a high amount of duplicated code.

Test Coverage (TC): Describes how well the MSA source code is tested. According to Bogner et al. [4] there are many different ways to determine the metric value. The most used way is the relative amount of covered lines and condition coverage based on the number of control flow branches.

5.2.3 Size

Definition: The size of an MSA is the aggregate size of all services. The large size has a negative impact on maintainability. The larger the aggregated size of all services in an MSA, the more difficult maintenance becomes. [3].

$$[MSA|SIZE] \xrightarrow{-} [Maintainability]$$

The Line of Code (LOC) describes the size of a service. However, the definition of size by LOC is controversial because it is challenging to infer the maintainability of a service from its size. However, the relative size ratio between services can be used to find potential services that are too large according to Shim et al. [29]. Therefore, the aggregated size of all services is not very meaningful for maintainability according to Bogner et al. [3].

There is only one metric $WSIC$ 5.2.1 to determine the size. However, we disagree with Bogner et al. that $WSIC$ can be used to determine the size of an MSA. According to Hirzalla et al. [11] the metric $WSIC$ is an indicator of the complexity rather than of the size of a service.

5.2.4 Complexity

Definition: Describes the required interaction of services to accomplish a task. That is the number of operations and tasks of a service and their direct and indirect use. High complexity has a negative impact on maintainability.

$$[MSA|COMPLEXITY] \xrightarrow{-} [Maintainability]$$

According to Bogner et al. [3] it is challenging to assign the existing metrics to complexity. The reason is that the authors of the existing metrics do not use unique categorizations. As a result, often no clear distinction is made between coupling, complexity, or size. In the following, we present two key metrics that we consider to best measure complexity.

Total Response for Services (TRS): $RFO(O)$ describes for an operation O the number of sequences of other operations and local methods that are executed when the operation O is called. $TRS(S)$ describes the sum of all Response for Operation $RFO(O)$ values for each operation of the Service Interface SI_S. Called services are counted as well [25].

$$TRS(S) = \sum_{O \in SI_S} RFO(O)$$

Number of Versions per Service (NOVS): $NOVS(MSA)$ describes the total number of versions over the total number of services within the MSA [11].

$$NOVS(MSA) = \frac{|V|}{|S|}$$

In an MSA used in production, it is not uncommon to support multiple versions of services to keep compatible with an older client application that uses the MSA. Increasing the number of versions per service makes the MSA difficult to maintain. According to Bogner et al. [3] it can be beneficial to set a threshold for both NVS_{AVG} and NVS_MAX.

5.2.5 Coupling

Definition: Describes the interdependence and interconnection of a service with other services. For example, an MSA composed by services which have just a few interconnections to each other can be maintained more easily. [3].

$$[MSA|COUPLING] \xrightarrow{-} [Maintainability]$$

The coupling can be easily analyzed by mapping the services links as a graph. This makes the coupling metric very clear and easy to define. In the literature, many different proposed metrics capture the coupling of services. In the following, a few of these metrics are presented.

Services Interdependence in the System (SIY): By Rud et al. [28], the metric SYI is defined. Here SYI represents the number of service pairs that call each other in opposite directions. As an example for a pair (S_1, S_2). The service S_1 calls an operation of the service S_2, and the service S_2, in turn, calls an operation of the service S_1. This type of attachment should be avoided in the context of MSA maintainability. Therefore, the ideal value for SIY is zero.

$$SIY(MSA) = \frac{|\{(S_1, S_2) with (S_1, S_2) \wedge (S_2, S_1) \in E_{MSA}\}|}{2}$$

Absolute Importance of the System (AIS): Rud et al. [28] define another metric AIS(S). The metric AIS(S) describes the number of services that invoke a service S. In a representation of the MSA as a graph; this is the number of incoming edges of the service S.

$$AIS(S) = \delta^-(V_S)$$

Bogner et al. [3] recommend not excluding clients that point to the same node. It is not helpful to set a threshold for the metric. Instead, the metric can be used to identify essential services.

Absolute Dependence of the Service (ADS): Similarly to AIS Rud et al. [28] propose ADS(S) to describe the number of services that are invoked by a service S. In a representation of the MSA as a graph, this is the number of outgoing edges of the service S.

$$ADS(S) = \delta^+(V_S)$$

Again it is not helpful to set a threshold for the metric. The metric can be used to identify impotent services [3].

Absolute Criticality of the Service (ACS): Rud et al. [28] defines the combination of AIS(S) and ADS(S) as followed:

$$ACS(S) = AIS(S) \times ADS(S)$$

The combination of AIS and ADS allows finding services with a high degree of coupling.

5.2.6 Cohesion

Definition: Cohesion describes how well individual parts or services within an MSA represent a logical task or unit. With a high cohesion, the maintainability of an MSA application can be improved. Unfortunately, cohesion is not extensively discussed in the literature. Only Perepletchikov et al. [24] have dealt with cohesion in detail. Three metrics of cohesion $SIDC$, $SIUC$, and $TSIC$ are presented below.

$$[MSA|COHESION] \xrightarrow{+} [Maintainability]$$

Service Interface Data Cohesion (SIDC) If all operations of the interface SI_S use the same parameter data types, the service S has a high cohesion defined as $SIDC(S) = 1$. To calculate the value for $SIDC(S)$, the number of operations with the same data types is divided by the total number of discrete data types for the interface SI_S. With $SIDC(S) = 1$ a maximum cohesion for a service is defined; thus a value close to 1 for SIDC(S) is necessary to ensure a good maintainability of S. [24] [3]

Service Interface Usage Cohesion (SIUC) The $SIUC(S)$ metric is used to describe how the calling behavior of clients is for the operations from the service interface SI_S. The service S is considered highly cohesive if every client uses every operation from SI_S. The value of the SIUC(S) metric thus describes the ratio between the number of operations used per client and the number of clients multiplied by the number of operations of the SI_S service interface. The value of the metric thus lies between 0 and 1. As with SIDC, a value close to 1 is an indicator of good serviceability of the service S. [24] [3]

$$SIUC(S) = \frac{\sum_{s \in S\{S\}} |(s, S)|}{\delta^-(V_S) * |O_S|}$$

Total Service Interface Cohesion (TSIC) TSIC is used to express the normalized sum of the two metrics SIDC and SIUC. The sum of the two metrics SIDC and SIUC is added and divided by two. [24] [3]

$$TSIC(S) = \frac{SIDC(S) + SIUC(S)}{2}$$

5.3 Portability metrics for MSAs

Portability is the other quality attribute that is investigated in our paper. However, as the results of the conducted SLR show, research on portability metrics for Service-Based architectures and especially Microservices, has been minimal. Based on the limited results, we offer a basic model for it. Direct Portability, Installability and Adaptability are factors of portability that are frequently mentioned in literature. This subsection of our paper investigates these quality factors. An overview of the relation of quality factors to the quality attribute portability is given in figure 4.

5.3.1 Direct Portability

Definition: Direct portability is the ability to take a piece of software directly and execute it on another platform without modification [17]. However, portability is not a binary quality. This type of metric has often been present in literature regarding portability [21] [17] [16].

$$[MSA|DirectPortability] \xrightarrow{+} [Portability]$$

Degree of Portability of the Service (DDPS): The metric is based on previous works regarding portability in software such as J. D. Mooney [21], and J. Lenhard [17]. It shows the degree to which a single Microservice could be ported from one environment to another. It can be expressed with the following equation:

$$DPS(S) = 1 - \frac{C_{port}(S, env1)}{C_{new}(S, env2)}$$

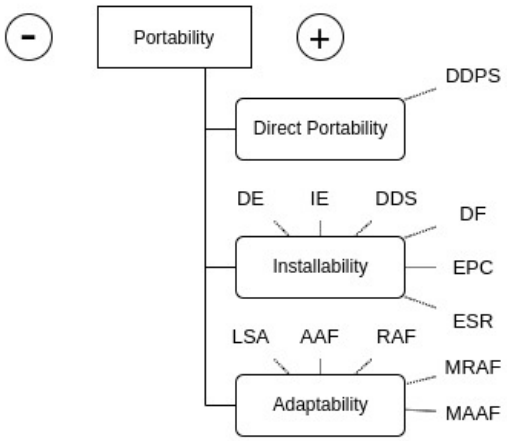

Figure 4: Quality Model for portability

C_{port} is the cost of changes that must be done on a service S to port it from an old environment $env1$ to a new environment $env2$, $C_{new}(S, env2)$ is the cost of rewriting a service S for a new environment $env2$. Assuming that the cost of porting will never be higher than the cost of rewriting, the degree of portability will always be between the range $[0, 1]$. If the degree of portability is equal to 0 then the service can be ported to a new environment without any changes, while if it is 1 represents a service that needs to be rewritten entirely. One thing to consider is how to measure the complexity. In studies, it is common to measure it using Lines of Code (LOC) [21] [17].

5.3.2 Installability

Definition: The ISO/IEC model defines installability as the "degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment" [13].

$$[MSA|Installability] \xrightarrow{+} [Portability]$$

According to the studies P11 and P13 from Lenhard et, al. [18] [16], installability can be subdivided into two sub-attributes, *Server Installability* and *Deployability*. In these studies, for each sub-attribute, a set of metrics that can measure it. For the rest of the installability subsection, the metrics from literature are presented.

a) Server Installability.

This sub-attribute of installability measures the ease of service installation, given that it is possible, and it is captured by the metrics *Ease of Setup Retry* and *Installation Effort* [18] [16].

Ease of Setup Retry (ESR): The metric measures how easy it is to repeat an installation successfully. Furthermore, it can be calculated as the ratio between the number of successful installations ($N_{success}$) and the total number of attempted installations (N_{total}).

$$ESR = \frac{N_{success}}{N_{total}}$$

Therefore $ESR \in [0, 1]$ where a value close to 1 signifies a failure-free installation process. Conversely, if the process is not bug-free or is prone to installation failures, then the value of ESR will be lower.

Installation Effort (IE): The metric measures the notion of difficulty of the installation process as the ratio between the *average time complexity (AIT)* and the *number of distinct steps (NDS)*. *NDS* counts the number of different operations that need to be performed for the installation process, and it is calculated using heuristic evaluation. *AIT* on the other hand, is calculated by running the steps found during the evaluation of NDS a suitable number of times and computing the average time required.

$$IE(S) = \begin{cases} 0 & \text{if NDS(S)} = 0, \\ \frac{AIT(S)}{NDS(S)} & \text{otherwise.} \end{cases}$$

b) Deployability.

The sub-attribute measures the work needed to deploy a service in a production environment. To deploy the service in the production environment, preparations must be done, including the packaging process and the construction of deployment descriptors [18] [16].

Effort of Package Construction (EPC): Measured by counting the number of folder structure creations (N_{fc}), descriptor creations (N_{dc}) and compression operations (N_{co}) that must be performed to construct the deployable executable.

$$EPC(S) = N_{fc} + N_{dc} + N_{co}$$

Deployment Descriptor Sizes (DDS): Measures the complexity of the descriptors needed for the deployment of a service and is calculated with the following formula:

$$DDS(S) = \sum_{i=1}^{N_{desc}} size(d_i)$$

Where N_{desc} is the total number of descriptors and $size(d_i)$ is calculated based on the type of the descriptor d_i (such as with *LOC* for plain text files or the number of elements for XML files).

Deployment Effort (DE): DE is an aggregation metric that combines the metrics EPC and DDS listed above. The metric is therefore calculated using the formula:

$$DE(S) = EPC(S) + DDS(S)$$

Deployment Flexibility (DF): Deployment can take different forms, which a server can support, and therefore it could influence deployability. The Deployment Flexibility metric counts the number of options for deployment that are supported by a server.

5.3.3 Adaptability

Definition: The ISO/IEC model defines Adaptability as "the degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments" [13].

$$[MSA|Adaptability] \xrightarrow{+} [Portability]$$

Metrics for adaptability corresponding explicitly to SBS, SOA, or MSA were not found. However, the study P12 offers metrics for measuring adaptability of a system at an architecture level [26]. The metrics in this paper are considered in terms of components of a system and services where

a set of components makes up the system, and these component require or offer services (functionalities/interfaces). The meaning of service in the original study differs from the meaning of a service used in the context of service-based systems. As such, the metrics from P12 are mapped to the context of Service-Based Systems. The main components of Service-Based Architecture and especially MSA are Services, and therefore components from the paper P12 are mapped to services. Additionally, we call as functions what P12 refers to as services.

Absolute adaptability of a function (AAF): The metric measures the number of services (denoted by US) used by an MSA for a given functionality f.

$$AAF(f) = |US_f|$$

Relative adaptability of a function (RAF): The metric measures how each functionality stresses its adaptability choices and gives a notion of how much more adaptable it could be by measuring the ratio between the number of used services to achieve the functionality over the number of services that offer such functionality.

$$RAF(f) = \frac{|US_f|}{|S_f|}$$

Mean of absolute adaptability of a function (MAAF): It is a metric that measures the mean of AAF. MAAF captures the notion of effort that is needed to manage each functionality.

$$MAAF = \frac{\sum_{i=1}^{n} AAF(i)}{n}$$

Mean of relative adaptability of a function (MRAF): It is a metric that measures the mean of RAF. MRAF gives an outlook on the mean utilization of the potential services for each function.

$$MRAF = \frac{\sum_{i=1}^{n} RAF(i)}{n}$$

Level of system adaptability (LSA): The metric measures the notion of adaptability for the entire system by evaluating the ratio of the number of services that compose the current system over the number of services that the most adaptable system would use. The level of system adaptability is captured by the following formula:

$$LSA = \frac{\sum_{i=1}^{n} AAF_i}{\sum_{i=1}^{n} |S_i|}$$

6. DISCUSSION

From the results of the SLR, research on Portability models is very limited compared to other quality attributes. When limiting the query string to MSA only, no results were found regarding portability. The results are minimal when broadening the scope of all service-based architectures. Our paper introduces a model of different sub-attributes of portability, such as installability and adaptability, based on different papers found by the SLR. However, for replaceability, another sub-attribute defined by the ISO/IEC standard [13], there were no results found. J. Lenhard has done intensive work regarding the portability of Process-Aware Systems on P13, where all the sub-attributes of portability are modeled in detail. However, most metrics introduced in the paper deal with process models and process languages (such as BMPN, BPEL). For the scope of this paper, a mapping could not be done between those metrics to the context of Microservices. However, this could be a possible area of future research.

Similarly, research on the maintainability quality model for MSA is also not extensive. However, Bogner et al. have laid a solid foundation by introducing and mapping maintainability models from object-oriented (OO) to SBS and SOA. Many of the metrics captured by Bogner et al. can be transferred to MSA. However, as with portability, these metrics have not been adequately researched. In particular, validation and testing of the metrics on real MSA are needed to verify the correctness of the metrics.

7. CONCLUSION AND FUTURE WORK

Research on quality assurance models of MSA has been very limited. There is no research that addresses the metrics of maintainability and portability for MSA. As such, the scope of research has been widened to considered SBSs and SOAs too. To answer RQ1, the current state of research on portability and maintainability was studied through a detailed systematic literature review. As an answer to RQ2 we propose a model of maintainability 3 based on the results of the literature review. This model is based on previous research on the topic. On the other hand, models of portability were non-existent when strictly speaking about Microservices. Therefore, we offer an initial simplistic model of portability 4 based on studies done in a larger scope and by mapping those results to MSA.

We also hope that these models will serve as a starting point for further research on this topic. Thus, there is a need to continue to advance the research and evaluate new metrics and formal definitions. In addition, empirical evaluation of the models in a real-world scenario is a possible area for future research.

8. REFERENCES

[1] D. R. Apolinário and B. B. de França. A method for monitoring the coupling evolution of microservice-based architectures. *Journal of the Brazilian Computer Society*, 27(1):17, Dec 2021.

[2] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. Limiting technical debt with maintainability assurance – an industry survey on used techniques and differences with service- and microservice-based systems. In *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 125–133, 2018.

[3] J. Bogner, S. Wagner, and A. Zimmermann. Automatically measuring the maintainability of service- and microservice-based systems: A literature review. In *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, IWSM Mensura '17, page 107–115, New York, NY, USA, 2017. Association for Computing Machinery.

[4] J. Bogner, S. Wagner, and A. Zimmermann. Towards a practical maintainability quality model for service-and microservice-based systems. In *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*, ECSA '17, page 195–198, New York, NY, USA, 2017. Association for Computing Machinery.

[5] E. Bouwers, J. P. Correia, A. v. Deursen, and J. Visser. Quantifying the analyzability of software architectures. In *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, pages 83–92, 2011.

[6] M. Cardarelli, L. Iovino, P. Di Francesco, A. Di Salle, I. Malavolta, and P. Lago. An extensible data-driven approach for evaluating the quality of microservice architectures. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, page 1225–1234, New York, NY, USA, 2019. Association for Computing Machinery.

[7] R. Ferenc, P. Hegedűs, and T. Gyimóthy. *Software Product Quality Models*, pages 65–100. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[8] B. Fluri, M. Wursch, and H. C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Proceedings of the 14th Working Conference on Reverse Engineering*, WCRE '07, page 70–79, USA, 2007. IEEE Computer Society.

[9] M. Fowler and J. Lewis. Microservices a definition of this new architectural term. `http://martinfowler.com/articles/microservices.html`, 2014. Retrieved May 20, 2022.

[10] H. Ghandorh, A. Noorwali, A. B. Nassif, L. F. Capretz, and R. Eagleson. A systematic literature review for software portability measurement: Preliminary results. In *Proceedings of the 2020 9th International Conference on Software and Computer Applications*, ICSCA 2020, page 152–157, New York, NY, USA, 2020. Association for Computing Machinery.

[11] M. Hirzalla, J. Cleland-Huang, and A. Arsanjani. *A Metrics Suite for Evaluating Flexibility and Complexity in Service Oriented Architectures*, page 41–52. Springer-Verlag, Berlin, Heidelberg, 2009.

[12] H. Hofmeister and G. Wirtz. Supporting service-oriented design with metrics. In *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, EDOC '08, page 191–200, USA, 2008. IEEE Computer Society.

[13] ISO/IEC 25010. Standard, International Organization for Standardization, Geneva, CH, 2011.

[14] B. A. Kitchenham, D. Budgen, and P. Brereton. *Evidence-Based Software Engineering and Systematic Reviews*, volume 4. Chapman & Hall/CRC Press, 2015.

[15] R. Koschke. Survey of Research on Software Clones. In R. Koschke, E. Merlo, and A. Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, volume 6301 of *Dagstuhl Seminar Proceedings (DagSemProc)*, pages 1–24, Dagstuhl, Germany, 2007. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[16] J. Lenhard. Portability of process-aware and service-oriented software: Evidence and metrics. 01 2016.

[17] J. Lenhard. *Improving Process Portability Through Metrics and Continuous Inspection*, pages 193–223. Springer International Publishing, Cham, 2017.

[18] J. Lenhard, S. Harrer, and G. Wirtz. Measuring the installability of service orchestrations using the square method. In *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*, pages 118–125, 2013.

[19] S. Li, H. Zhang, Z. Jia, C. Zhong, C. Zhang, Z. Shen, and M. A. Babar. Understanding and addressing quality attributes of microservices architecture: A systematic literature review. *Information and Software Technology*, 131, March 2021.

[20] M. Loukides and S. Swoyer. Microservices Adoption in 2020. `https://www.oreilly.com/radar/microservices-adoption-in-2020/`, 2020. Retrieved July 5, 2022.

[21] J. D. Mooney. Issues in the specification and measurement of software portability. 2001.

[22] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis. Microservices in practice, part 1: Reality check and service design. *IEEE Software*, 34(1):91–98, 2017.

[23] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis. Microservices in practice, part 1: Reality check and service design. *IEEE Software*, 34(1):91–98, 2017.

[24] M. Perepletchikov, C. Ryan, and K. Frampton. Cohesion metrics for predicting maintainability of service-oriented software. In *Seventh International Conference on Quality Software (QSIC 2007)*, pages 328–335, 2007.

[25] M. Perepletchikov, C. Ryan, K. Frampton, and Z. Tari. Coupling metrics for predicting maintainability in service-oriented designs. In *Proceedings of the 2007 Australian Software Engineering Conference*, ASWEC '07, page 329–340, USA, 2007. IEEE Computer Society.

[26] D. Perez-Palacin, R. Mirandola, and J. Merseguer. On the relationships between qos and software adaptability at the architectural level. *Journal of Systems and Software*, 87:1–17, 2014.

[27] M. Richards. *Microservices vs. Service-Oriented Architecture*. O'Reilly Media, Inc., April 2016.

[28] D. Rud, A. Schmietendorf, and R. Dumke. R.: Product metrics for service-oriented infrastructures. 01 2006.

[29] B. Shim, S. Choue, S. Kim, and S. Park. A design quality model for service-oriented architecture. In *Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*, APSEC '08, page 403–410, USA, 2008. IEEE Computer Society.

APPENDIX

Table 1: Reviewed Studies

Code	Study Title	Ref.
P1	Towards a Practical Maintainability Quality Model for Service and Microservice-based Systems	[4]
P2	Automatically Measuring the Maintainability of Service- and Microservice-based Systems – a Literature Review	[3]
P3	Limiting Technical Debt with Maintainability Assurance – An Industry Survey on Used Techniques and Differences with Service- and Microservice-Based Systems	[2]
P4	An Extensible Data-Driven Approach for Evaluating the Quality of Microservice Architectures	[6]
P5	A method for monitoring the coupling evolution of microservice-based architectures	[1]
P6	A Design Quality Model for Service-Oriented Architecture	[29]
P7	Supporting Service-Oriented Design with Metrics	[12]
P8	A Metrics Suite for Evaluating Flexibility and Complexity in Service Oriented Architectures	[11]
P9	Coupling Metrics for Predicting Maintainability in Service-Oriented Designs	[25]
P10	Product Metrics for Service-Oriented Infrastructures	[28]
P11	Measuring the Installability of Service Orchestrations Using the Square Method	[18]
P12	On the relationships between QoS and software adaptability at the architectural level	[26]
P13	Portability of process-aware and service-oriented software: Evidence and metrics	[16]

Correlation and causation between Technical Debt and Quality

Shu Zhang
RWTH Aachen University
shu.zhang@rwth-aachen.de

Lukas Jansen
RWTH Aachen University
lukas.maximilian.jansen
@rwth-aachen.de

ABSTRACT

Technical Debt is a metaphor to describe "not-quite-right" code. This debt can in most cases speed up software development in the short term, but at the risk of paying a higher cost later.

Therefore, in this paper, we will examine the correlation and causation between Technical Debt and quality. We attempt to find several models to help us measure Technical Debt and quality. With these models and methods, we continue our analysis of what aspects of software quality are positively or negatively affected by Technical Debt.

We can find a clear correlation between Technical Debt and software quality for the similar characteristics they have in many aspects. When the Technical Debt reaches a threshold, it may slow down the software development, which is similar to what many software developers would expect. At this threshold, more time is spent on working around Technical Debt, which can be called paying interest on that debt than is gained by speeding up development by taking on Technical Debt. This can negatively affect software quality. But on the other hand, if developers can manage Technical Debt in suitable ways, and never reach the threshold, Technical Debt can also enhance the efficiency of software development, thus allowing for more time to improve software quality. There also has to be a strategy for paying back the debt. This can be a gradual process where instead of working on new features, developers pay back some of the debt every so often.

The results show that Technical Debt can have a varied impact on software quality. Taking on Technical Debt is not necessarily a bad idea, but the debt has to be managed correctly.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*productivity, programming teams, software configuration management*

SWC Seminar 2022 RWTH Aachen University, Germany.

Keywords

Software Quality, Technical Debt, ISO 9126, SQALE Method

1. INTRODUCTION

Before considering the topic at hand, the aspects of Technical Debt (TD) and quality are introduced respectively.

1.1 What is Technical Debt

TD is a metaphor introduced by Ward Cunningham to describe the effects of deficiencies in internal quality to non-technical product stakeholders.

Writing "not-quite-right code" [10] is like going into debt. It can speed up development initially, but additional time spent on this suboptimal code counts as interest on that debt [10]. When there is TD, developers have to put in extra time and effort to continually fix the problems and side effects caused by previous compromises. While there may seem to be immediate benefits to be gained, software engineers may have to pay that debt back in the future. Rewriting the code can be interpreted as the debt being paid back.

A codebase containing a plethora of TD is hard to maintain and can make the product inflexible, and the whole department may have problems due to loose implementation or incomplete object-oriented design.

While the metaphor of TD is based on financial debt, there are some differences from financial debt. One is that the debt is not necessarily taken on by the same person that has to pay it back. This is because developers often don't have to maintain the code that they have written themselves when it goes into the maintenance phase and is no longer actively developed [4]. Often debt does not have to be paid back at all. When a product is retired, all the existing debt is ignored and does not have to be paid back, unlike financial debt.

1.2 What are the aspects of Software Quality

In general, regarding Software Quality, there are external quality and internal quality. The external quality is usually considered from the view of users. For example, whether the software functions correctly, the performance of the software, the usability, and the security of the software. The internal quality is considered from the view of developers, and it the concerns readability of the code, ease of maintenance, good scalability, reusability, and ease of testing.

External quality is the perceived usefulness of a system. It provides users with value and meets the specifications of the software developers. This quality can be measured through functional testing, quality assurance, and user feedback. It

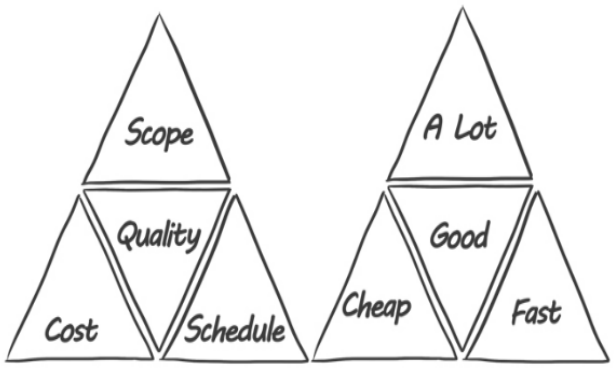

Figure 1: Golden Triangle [1]

is the quality that directly affects the users.

Ease of maintenance is how easy it is to change parts of the code in the future. Scalability is how easy it is to increase the functionality of the product[11]. Reusability is the degree to which an asset can be used in more than one software system, or in building other assets[22].

1.3 Motivation

In the research there are various ways to describe the boundaries of project quality. The simplest of these is 'The Golden Triangle' (Figure 1)which includes four dimensions [1]. The reason why quality is placed in the middle of the triangle is that quality is a reflection of the result of balancing the other three factors. For example, if the scope is not reduced, the cost is not increased, and if the team wants to save time and take shortcuts, the quality will be affected, and this quality is not only the quality of the product but also the quality of the architecture and the quality of the code.

In this paper the authors want to answer the following research questions:

RQ1 How can TD positively affect software quality?

RQ2 How can TD negatively affect software quality?

2. RELATED WORK

There is a lot of existing research on TD and quality. Many people focus on the management of and models of both. These models exist the help stakeholders understand these abstract concepts. Therefore, the focus is on the two concepts and to provide the most known models.

2.1 Software Quality

Software is the key to the development process of many organizations and is becoming an integral part of human life [22]. This is why the quality of software products is now considered an important factor in commercial success [3]. It is defined by the IEEE [1990] as the *degree to which a system, component, or process meets specified requirements and customer (user) needs (expectations).* And Quality assurance, as defined by TechTarget, establishes and maintains the requirements for developing or manufacturing reliable products.

Software quality is important for developers and users. Developers want a successful product that will continue to run without crashing and run reliably, free of bugs and defects. A high-quality product means a competitive product on the market. Ensuring that all products are up and running can enhance the reputation of an organization. With good software quality, the company can attract good developers, because they should aspire to create a culture of technical excellence, where quality is not negotiable. A quality assurance system is designed to increase customer confidence and the company's reputation while improving workflow and efficiency and enabling the company to better compete with others [14].

2.1.1 McCall's Quality Model

McCall's Quality Model[20] presented in 1977 is one of the most well-known quality models in software engineering history. McCall's model was developed by the US air-force electronic system decision (ESD), the Rome air development center (RADC), and general electric [25], for the quality of software.

McCall's model established software quality through three aspects: Product Revision, Product Operation, and Product Transition. Every quality factor has a set of quality criteria, and every quality criterion could be reflected by one or more metrics [3]. The contents of McCall's Quality Model are the following:

Product Revision It is about the ability of the product to change.

- Maintainability: The quantity of work required to diagnose and modify a running software to meet new user requirements, or when the environment changes or new errors are found during the operation.
- Flexibility: The quantity of work required to modify or improve a piece of software that is already in operation.
- Testability: The quantity of work required to test the software to ensure it can perform its intended function.

Product Operations It is about adaptability to new environments.

- Correctness: The ability to which the software meets the design specification and the user's intended objectives in the intended environment. The goal is that it is going to require that the software is free of errors.
- Reliability: The ability to which the software continues to operate without failure for a specified period of time and under specified conditions, under the design requirements.
- Efficiency: The number of computer resources required by the software system to perform the intended function.
- Integrity: The ability to protect data from accidental or intentional destruction, alteration, or loss for a particular purpose.
- Usability: The quantity of work required for a software system for users to learn, use the software and prepare inputs and interpret outputs for the program.

Product Transition It is about the basic operational characteristics.

- Portability: The quantity of work required to port a software system from one computer system or environment to another computer system or environment.

- Reusability: The ability of a software (or a component of software) to be reused in other applications (the functionality of which is related to the extent to which the software or software component performs the function.

- Interoperability: The quantity of work required to connect software to other systems. If this software is to be networked or communicate with other systems or bring other systems under its control, there must be interfaces between the systems to allow them to be linked.

In more detail, McCall's quality model consists of 11 quality factors that describe the external perspective of the software, that is, the customer or user perspective. 23 quality criteria that describe the internal perspective of the software, that is the developer perspective, and a set of metrics that define and are used to provide standards and methods of measurement. The key to this model is the relationship between quality characteristics and metrics. However, the model does not directly consider the functionality of the software [2].

2.1.2 Boehm Model

In 1978, B.W. Boehm developed his software quality model. The model represents a hierarchical quality model which is similar to the McCall quality model, using a set of predefined attributes and metrics to describe software quality. These attributes contribute to the total software quality [9].

Boehm adds new factors to McCall's model and emphasizes the maintainability of software products. This model aims to address contemporary models' shortcomings which automatically assess software quality quantitatively [2].

The Boehm Model structure develops three levels of characteristics, namely High-Level Characteristics, Intermediate-Level Characteristics, and Primitive Characteristics. Each High-Level Characteristic contains many Intermediate-Level Characteristics, and Intermediate-Level Characteristics also contain corresponding Primitive Characteristics. In Boehm Model, the High-Level Characteristic represents the basic high-level requirements for practical use, into which the evaluation of software quality can be placed. There are three high-level characteristics, namely As-is Utility, Portability, and Maintainability [9]. Next, the Intermediate-Level Characteristics in each High-Level Characteristic are illustrated.
As-is utility Extent to which, the software product can be used as-is.

- Reliability: The ability to which the software continues to operate without failure for a specified period of time and under specified conditions, under the design requirements.

- Efficiency: The number of computer resources required by the software system to perform the intended function.

- Human Engineering is similar to Usability: The quantity of work required for a software system for users to learn, use the software and prepare inputs and interpret outputs for the program, gets more focused on Communicativeness.

Portability Effort required to change when the environment has been changed. Portability has no Intermediate-Level Characteristics, but it connects directly to Device Independence and Self-Containedness.

- Device Independence: The ability of a software product to co-exist in a public environment with other independent software with which it shares public resources.

- Self-Containedness: The ability of a software product to replace another software product for the same purpose in the same environment.

Maintainability Effort required to detect and fix an error of the software product.

- Testability: The quantity of work required to test the software to ensure it can perform its intended function.

- Understandability: The ability of a software product to enable users to understand whether and how the software is suitable and can be used for specific tasks and conditions of use (documentation, initial impressions of functionality).

- Modifiability: Software products that give users the ability to operate, control, and modify it.

Primitive Characteristics can be used to provide the basis for defined quality metrics, many of which are highly similar to McCall's quality model, but the Boehm Model is more focused on the Operability of users and tries to satisfy the needs of users. And Boehm defined the 'metric' as a measure of the extent or degree to which a product possesses and exhibits a certain (quality) characteristic [9]. Boehm's model is an improvised version of McCall's quality model, but it also has its weaknesses, which continue to manifest themselves in the course of its development.

2.1.3 A generic standard ISO 9126

As a large number of software quality models were developed, confusion occurred and new standard models were needed. As a result, ISO/IEC JTC1 began to develop the required convergence and encourage worldwide standardization. [2] In 1991, the ISO published its first international consensus about the quality characteristics of software. It reflects the sum of features and characteristics of a software product's ability to satisfy specified and potential requirements [3].

As can be seen in Figure 2, this model is a hierarchical tree structure composed of characteristics and sub-characteristics. The highest level of this structure consists of quality characteristics and the lowest level consists of software quality criteria. The model defines six characteristics, including functionality, reliability, usability, efficiency, maintainability, and portability. These characteristics are further classified into 21 sub-characteristics [8].

In the ISO 9126 Quality Model, internal quality attributes have an impact on external quality attributes, while external attributes also have an impact on in-use quality attributes. In addition, in-use quality depends on external quality, while external quality depends on internal quality [3].

Figure 2: The contents of ISO 9126 Quality Model[17]

When comparing ISO 9126 Quality Model with McCall's quality model and Boehm Model, it is clear that McCall's quality model and Boehm Model are the basis of the ISO 9126 Quality Model and that all the characteristics in ISO can be referred to in both of the above models. So it can be said that ISO 9126 is a general standard quality model, but its separate internal and external quality measures are its shortcoming.

2.2 Technical Debt

Ever since the coinage of the term by Cunningham in 1992, TD has been of large research interest, both in academia and the industry itself. This is even though TD is not directly visible to the customer of a product, compared to software quality.

After first giving an ontology and categories of TD, ways to measure and find TD are covered.

2.2.1 An Ontology of Terms on Technical Debt

TD is can be divided into multiple different types of debt. These are Architecture Debt, Build Debt, Code Debt, Defect Debt, Design Debt, Documentation Debt, Infrastructure Debt, People Debt, Process Debt, Requirement Debt, Service Debt, Test Automation Debt, and Test Debt. These types of debt have different indicators by which TD can be measured[5].

For example code debt is illegible source code which can be indicated by duplicated code, non-standard-conforming code, or inefficient algorithms.

Source code comments can contain information about a special type of TD: Self-admitted TD [19]. This debt fits into the deliberate section in Fowler's quadrant because the developer knew that something was "not-quite-right" [10] and because of that mentioned it in a comment. But not all categories from there can be found in source code comments. E.g. infrastructure debt can not be identified from the source code. The five categories which can be identified are found by the following types of comments [5].

Design debt Comments mentioning that the code is misplaced, suffers from a lack of abstraction, is poorly implemented, a workaround, or a temporary solution.

Defect debt When the comment contains information about a known defect in the code.

Documentation debt Missing or incomplete documentation is mentioned.

Requirement debt The incompleteness of the code is mentioned.

Test debt Comments that state that tests are nonexistent, or need to be improved.

2.2.2 The categories of Technical Debt

Martin Fowler defined a quadrant of four classifications of TD. This quadrant groups TD according to how and why TD was taken.

Another categorization is between self-admitted TD and not self-admitted TD.

	Reckless	Prudent
Deliberate	'We do not have time for design'	'We must ship now and deal with consequences'
Inadvertent	'What is Layering?'	'Now we know how we should have done it'

TD Quadrant[13]

Reckless Deliberate
This quadrant reflects a team that takes shortcuts without design and does not follow good development practices because of cost and time, has no follow-up plans for TD. For example, codes directly without design and has no intention of refactoring the code later.

Prudent Deliberate
This quadrant reflects a situation where the team is clear about the benefits and consequences of the TD and also has a follow-up plan to improve the architecture and enhance the quality of the code. For example, to release the product as soon as possible, they develop it in a rushed way first and then refactor the code, this kind of debt can be repaid in time so that it can have some short-term benefits and no negative impact on the long term.

Reckless Inadvertent
This quadrant reflects a situation where the team is unaware of the TD and does not know that it has to be subsequently repaid. For example, the development teams are ignorant of architectural design, and their code is thus very unstructured.

2.2.3 Short-Term and Long-Term Technical Debt

McConnell [21] proposes the distinction between short-term and long-term debt. Short-term debt is taken on tactically and reactively. This can be the case when due to unforeseen problems shortly before a planned release, it is decided to use a quick and maybe not perfect solution to these problems. It is expected that this kind of debt is then paid back quickly. Long-term debt on the other hand is taken on strategically and proactively [21]. That means there does not have to be a reason like the problems described above to take on debt. Instead, it is decided that taking on the debt will have a long-term benefit and it is not expected to be paid back quickly. One example of long-term TD can be when a product is quickly developed, without much regard to TD, just reach the market as fast as possible [21]. Otherwise, the product could appear too late which would result in lost revenue, or maybe a competitor has already established their product.

Prudent Inadvertent
This quadrant reflects the fact that the team takes architectural design and TD seriously, but TD arises due to changes in the business, or other objective factors. For example, if during the initial design the team cannot accurately predict the development of the following business, and as the business grows, the design cannot meet the new requirements, it is difficult to avoid this TD in this way, but if the architecture can be upgraded and refactored promptly, it can be ensured that there will be no serious impact.

2.2.4 SQALE Method

SQALE, short for Software Quality Assessment Based on Lifecycle Expectations, is a method to evaluate the source code of a software application. It is compromised of the following four key concepts [18]. This evaluation can show TD in the source code.

The Quality Model Used for formulating and organizing the non-functional requirements that relate to code quality.

The Analysis Model Rules for normalizing the measures and the violations relating to the code as well as rules for aggregating the normalized values.

The Indices Represent the cost, used to represent and analyze the TD.

The Indicators Three summarized indicators are used to provide a visual representation of the TD.

The SQALE is specifically adapted to consider TD in addition to software quality.

2.2.5 Repair Effort

Repair Effort (RE) is the cost of bringing the internal software quality back to the ideal level after taking on TD[23]. The RE consists of the Rework Fraction (RF), the Rebuild Value (RV), and the Refactoring Adjustment (RA). This repair effort can be specified in man-months and is the product of RF, RV, and RA. This relationship can be represented by the following formula:

$$RE = RF \cdot RV \cdot RA$$

Target/Source	1-star	2-star	3-star	4-star	5-star
1-star					
2-star	60%				
3-star	100%	40%			
4-star	135%	75%	35%		
5-star	175%	115%	75%	40%	

Figure 3: Estimated Rework Fraction[23]

RF is the percentage of lines that need to be changed to increase the quality of the code from one level to another. E.g. if the quality is already four out of five only an estimated 40% of lines have to be changed, because the quality is already quite good. Similarly, if the quality is very poor (one out of five) 175% of the lines have to be changed. Figure 3 shows this relationship.
RV is the number of man-months that it takes to rebuild a system. RV depends on the size of the system and on the technology that is used to build the system.
RA represents the experience and tooling a team has which increases the team's productivity when refactoring. A team with lots of experience can take 10% less time, which is represented by the RA value.

2.2.6 Static code analysis

Another way the measure TD is through static code analysis [6]. Many tools can analyze a project's source code to give insights into possible problems that may result in faults or just TD. In the case of TD, most of them give the principal of the debt in time to remove issues [6], which is like the repair effort described above. Some may also calculate the interest on the debt [6]. Because of the way these tools work they are limited to only architectural, design, and code debt from the ontology of terms on TD.

3. METHODOLOGY

The goal of our study was to investigate the correlation and causation between TD and software quality. To get the result a literature review in the domains of TD and software quality was conducted. For finding the relevant literature *Google Scholar* was used. The search queries were *TD* and *Software quality* and also in conjunction with other relevant words like *model*, *measure*, or *management*. From these papers, their sources were also considered to find further papers as well.

The authors primarily looked for papers included in journals and conferences and therefore most of the resulting papers were from the *IEEE* or *ACM* databases. But some sources were not from academia but the industry instead. These were often just websites but still sound based on the fact that they were also cited in the academic papers. The relevance of the papers was based on whether they might provide an answer to the research question. Although the authors searched in recent literature, the foundation which is in parts quite old was also used.

Both authors evaluated the papers on their own and then combined the results. For some papers where only a part of them is relevant to our research question, only these parts were evaluated and not the paper in its entirety.

4. RESULTS

Characteristics	ISO 9126	SQALE Method
Functionailty	×	
Reusability	×	×
Portability	×	×
Maintainability	×	×
Security		×
Efficiency	×	×
Changeability		×
Reliability	×	×
Testability		×

Figure 4: The similar characteristics between ISO 9126 and SQALE Method

Many software quality models were found in the literature. To keep everything consistent, the authors decided to only focus on one quality model in the results. ISO 9126 is based on McCall's Quality Model and Boehm Model. The model has two main parts of the attributes of internal and external quality and the quality in use attributes. Moreover, ISO 9126 has been used as the basis for Tailored Quality Models. One of its features standardizes software quality [22]. According to our analysis ISO 9126 is the most representative and typical software quality model. The SQALE is a software quality Assessment that is based on lifecycle expectations. It provides a comprehensive assessment of the TD of the software source code [18].

While causation and correlation can exist at the same time, correlation does not imply causation, but causation always implies correlation. In the introduction to models of software quality and TD above, some minor causation exists in the correlation between software quality and TD. The authors try to explain this causation by way of studying how can TD positively or negatively affect software quality.

4.1 The similar characteristics between ISO 9126 and SQALE Method

As can be seen from Figure 4, the five characteristics are similar in both ISO 9126 Quality Model and SQALE Method, are Reusability, Reliability, Efficiency, Maintainability, and Portability.

As mentioned above, Testability and Changeability are sub-characteristics of Maintainability in the ISO 9126 Quality Model. But in the SQALE Method, they are considered an early characteristic in the life cycle, so Testability and Changeability become the characteristics instead of sub-characteristics [18]. The definition of Testability in ISO 9126 is *the capability of the software product to enable modified software to be validated* [3]. At the same time, SQALE Method has made Regulations for the Testability of software: There is no method with a cyclomatic complexity over 12 [18]. Reducing the cyclomatic complexity of the software also allows developers to modify the software much easier. Furthermore, Changeability is defined by ISO 9126 as *the capability of the software product to enable a specified modification to be implemented* [3]. SQALE Method contains a requirement for Changeability. For example, there is no cyclic dependency between packages [18]. The cyclic dependency between packages a dependency can cause problems with another package when a package is modified for a specific function of the software. No cyclic dependency also ensures that developers have access to implement a specified modification to the software.

Security is also one of the important characteristics in the SQALE Method, although it does not appear at the first level of the ISO 9126 Quality Model because it is a sub-characteristic of Functionality. Being a sub-characteristic should not imply it is irrelevant to software quality. Research shows[16] that Functionality is acknowledged as the most important aspect of software system quality in Safety-critical Domains. For Security, ISO 9126 states that *it is the capability of the software product to protect information and data so that unauthorized persons or systems cannot read or modify them and authorized persons or systems are not denied access to them* [3]. An index called SQALE Security Index (SSI) is specified in the SQALE Method [18]. This represents a component of the TD of the source code being assessed. The index also guarantees the refusal of unauthorized individuals.

4.2 How Technical Debt can positively affect software quality

If the debt is managed correctly by for example only being taken on for a short duration and then promptly repaid, it can also be beneficial, by allowing, for example, faster shipping of the product. In this case, the debt will not have a negative impact on most aspects of the software quality. Maintainability, reusability, and testability would still suffer from this short-term debt, at least temporarily. This is because the TD makes working on that part of the source code harder, thus negatively affecting its maintainability and reusability. Testability could also suffer from this because of an overly complex solution or because a unique solution has to have a special test for it. Therefore the debt should be repaid quickly to not hinder development in the future too much [4]. This should not be done excessively because TD works differently than financial debt [12]. If the product is near the end of its life the debt does not matter because there will be very few interest payments and when the product is retired the remaining debt is no longer a problem [4]. While the debt is not repaid, because nothing was done to fix the TD, instead this could be seen as the debt being forgiven. This should not be used to justify taking on lots of debt late in a product lifecycle because some of the code could be reused in a future product if it would not contain much or any TD. This reusability can then improve the quality of future products because many issues regarding the reused code have already been fixed. That is because eventual problems have already appeared while the product was in use and are already fixed.

Wehaibi et al. [27] have found that SATD, which is part of the deliberate debt from Fowler's debt quadrant [13], in a source code file does not lead to more defects in that file. Therefore this type of TD does not have an impact on many of the categories under Product Operation from McCall's Quality Model [20]. The opposite can be the case because new features can be developed more quickly when taking on TD [4]. There should however be a distinction between short-term and long-term debt that McConnell [21] also proposes. Short-term debt can be beneficial for new features while long-term debt can cripple the development team so

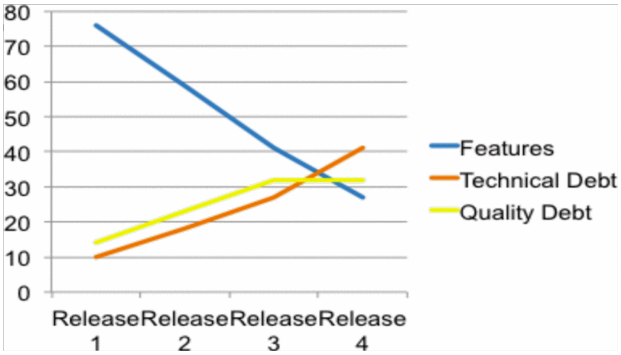

Figure 5: Impact of low investment in debt reduction over multiple releases [24]

much, that they can produce new features and instead their time is taken up by paying interest on potentially lots of very old TD [4].

Due to the nature of TD, in the systems that are not modified, but provide the expected level of service, paying back that debt could be deemed an additional unnecessary cost. It could also be decided that these parts of the systems are built intentionally containing TD. Then the total cost at the end of the product's lifecycle might still be lower than if they were built without any TD. Although the quality of software could increase due to refactoring, it might also happen that new bugs negatively affecting the quality would be introduced. Changing these parts of the source code that contain TD but are stable and not often changed, may also introduce new bugs [7]. These new bugs then decrease the functionality and reliability of the product. This way maintainability may be traded for potentially worse functionality and reliability. Because of this leaving the TD as is can benefit the software quality by not potentially decreasing it through a bug introducing refactoring of the source code.

Another kind of system where TD can positively affect software quality is internal tools. Because the end-user of the product will never interact with them the maybe low software quality of the tools themselves does not affect the quality of the product. The opposite is the case because the increased developer productivity offered by these tools can then improve the software quality of the product.

4.3 How Technical Debt can negatively affect software quality

If lots of debt accumulates it can make the software product very hard to maintain[12]. Due to maintainability being a quality characteristic of software quality, for example in the ISO-9126 quality model [17], this leads to a decrease in quality.

Although the authors consider that TD may bring short-term benefits to a project, such benefits often come at the cost of additional work in the future, similar to paying interest on a debt [15]. Significant TD affects the productivity of software development teams and reduces their morale and motivation. This is the case when more time has to be spent working around that TD than is gained by taking on that debt and speeding up development this way. The accumulation of TD leads to a vicious circle: low productivity causes managers to roll out more features and leads to delays in TD issues, which turn to a further increase in TD. This low morale can also lead to a high rate of turnover [26], which only makes the situation worse because developers familiar with the project will leave. That will then lead to an even less productive team, or worse even a decrease in quality because the team is experienced overall. According to the research findings of Ken Power [24], TD will have a negative impact on the software team's velocity of functionality if it has been ignored, which will severely compress software development time. A software team may ignore TD for the first release, or even the second release. The data in Figure 5 shows the impact of four consecutive releases in a typical case, and as TD rises, the team's feature velocity falls rapidly. More time is being devoted to reducing TD to deal with the growing backlog of debt items. In extreme cases teams may be forced to spend almost the complete release lifecycle on reducing TD. Consequently, no time will be spent on improving quality [24]. A high level of TD means that a lot of effort is allocated to maintenance. This will ultimately lead to a bankrupt software team and no guarantee of software quality. Bankruptcy works similarly to financial debt because at that point the software team can either only focus on paying back the principal and thus ceasing all feature development or do a complete rewrite. Not only does it waste a lot of time, but it also completely affects the Maintainability and Efficiency of software.

5. CONCLUSIONS

In this paper, we first researched the definitions of TD and quality in the literature. With these findings, we then analyzed the relationship between TD and software quality. In particular, what aspects of software quality can benefit or are negatively affected by TD.

We found out that there is a strong correlation between TD and quality. This is because TD and quality share similarities in many aspects. Most of the time TD will have a negative impact on quality, mainly on the aspects of maintainability and reusability. Also, if a certain threshold of debt is reached all development slows down and no new features can be implemented at all. The opposite however can also be the case if the TD is managed correctly and this critical threshold is never reached. When TD is only taken on for a small amount of time it can greatly increase development speed. Therefore TD can be beneficial to software quality by improving the functionality of the product.

TD will just be forgotten when a product is retired and thus will possibly never have to be repaid. If a part of the source code contains TD but is never or rarely changed, the debt also does not have a negative impact. The developers that introduce TD into the source code will often not be negatively affected by it themselves when they are assigned to another project or when they leave. This gives them an incentive to not care about TD as much as they should.

Despite possible benefits of taking on TD, we found that developers should keep TD to a minimum because it possibly has a negative impact on the software product.

6. FUTURE WORK

We found lots of theoretical work on the effects of TD on software quality, but there are very few case studies or empirical work that look into these effects in practice. Such a practical evaluation could help validate the theoretical approach to TD. One possible way to conduct such a study

would be to use a static code analysis tool on parts of an existing codebase but not its entirety. Half of the files would be analyzed using the tool while the other half would not. The developers should then remove all the TD found by this analysis. Then after some time the number of faults and bugs per file which were analyzed compared to files that were not analyzed could give an estimation of the impact of TD on software quality in practice. E.g. if the files containing no TD according to the analysis tool also lead to fewer faults then TD would have a negative impact on software quality in practice.

7. REFERENCES

[1] The golden triangle. `https://beingaprojectmanager.com/nuggets/project-management-golden-triangle/`. Accessed: 2017-08-14.

[2] A. B. Al-Badareen, M. H. Selamat, M. A Jabar, J. Din, and S. Turaev. Software quality models: A comparative study. In *Software Engineering and Computer Systems*, pages 46–55, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[3] R. Al-Qutaish. Quality models in software engineering literature: An analytical and comparative study. volume 6, 11 2010.

[4] E. Allman. Managing technical debt: Shortcuts that save money and time today can cost you down the road. *Queue*, 10(3):10–17, mar 2012.

[5] N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola. Towards an ontology of terms on technical debt. In *2014 Sixth International Workshop on Managing Technical Debt*, pages 1–7. IEEE, 2014.

[6] P. C. Avgeriou, D. Taibi, A. Ampatzoglou, F. Arcelli Fontana, T. Besker, A. Chatzigeorgiou, V. Lenarduzzi, A. Martini, A. Moschou, I. Pigazzini, N. Saarimaki, D. D. Sas, S. S. de Toledo, and A. A. Tsintzira. An overview and comparison of technical debt measurement tools. *IEEE Software*, 38(3):61–71, 2021.

[7] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo. When does a refactoring induce bugs? an empirical study. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 104–113. IEEE, 2012.

[8] B. Behkamal, M. Kahani, and M. K. Akbari. Customizing iso 9126 quality model for evaluation of b2b applications. *Information and software technology*, 51(3):599–609, 2009.

[9] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, pages 592–605, 1976.

[10] W. Cunningham. The wycash portfolio management system. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '92, page 29–30, New York, NY, USA, 1992. Association for Computing Machinery.

[11] R. G. Dromey. A model for software product quality. *IEEE Transactions on software engineering*, 21(2):146–162, 1995.

[12] M. Fowler. Technical debt. `https://martinfowler.com/bliki/TechnicalDebt.html`. Accessed: 2022-05-25.

[13] M. Fowler. Technical debt quadrant. `https://martinfowler.com/bliki/TechnicalDebtQuadrant.html`. Accessed: 2022-05-12.

[14] A. S. Gillis. Quality assurance. `https://www.techtarget.com/searchsoftwarequality/definition/quality-assurance/`.

[15] Y. Guo, R. O. Spínola, and C. Seaman. Exploring the costs of technical debt management–a case study. *Empirical Software Engineering*, 21(1):159–182, 2016.

[16] F. Huang, Y. Wang, Y. Wang, and P. Zong. What software quality characteristics most concern safety-critical domains? In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 635–636, 2018.

[17] International Standard Organization (ISO). International standard iso/iec 9126, information technology - product quality - part1: Quality model, 2001.

[18] J.-L. Letouzey. The sqale method for evaluating technical debt. In *2012 Third International Workshop on Managing Technical Debt (MTD)*, pages 31–36, 2012.

[19] E. d. S. Maldonado and E. Shihab. Detecting and quantifying different types of self-admitted technical debt. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pages 9–15, 2015.

[20] J. A. McCall, P. K. Richards, and G. F. Walters. Factors in software quality. volume i. concepts and definitions of software quality. Technical report, GENERAL ELECTRIC CO SUNNYVALE CA, 1977.

[21] S. Mcconnell. Managing technical debt. *Construx*, pages 1–14, 01 2013.

[22] J. P. Miguel, D. Mauricio, and G. Rodríguez. A review of software quality models for the evaluation of software products. *arXiv preprint arXiv:1412.2977*, 2014.

[23] A. Nugroho, J. Visser, and T. Kuipers. An empirical model of technical debt and interest. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, MTD '11, page 1–8, New York, NY, USA, 2011. Association for Computing Machinery.

[24] K. Power. Understanding the impact of technical debt on the capacity and velocity of teams and organizations: Viewing team and organization capacity as a portfolio of real options. In *2013 4th International Workshop on Managing Technical Debt (MTD)*, pages 28–31, 2013.

[25] T. Ravichandran and M. A. Rothenberger. Software reuse strategies and component markets. *Communications of the ACM*, 46(8):109–114, 2003.

[26] E. Tom, A. Aurum, and R. Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516, 2013.

[27] S. Wehaibi, E. Shihab, and L. Guerrouj. Examining the impact of self-admitted technical debt on software quality. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 179–188, 2016.

Exploring the Relation Between Technical Debt and Risk Management

Katharina Güths
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
katharina.gueths@rwth-aachen.de

Egzon Ademi
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
egzon.ademi@rwth-aachen.de

ABSTRACT

Technical debt is a metaphor in software engineering that describes the acceptance of future costs in favor of short-term benefits. Possible consequences are manifold and include additional financial costs as well as quality issues and reduced maintainability. Nevertheless it is a common phenomenon in software which can emerge unnoticed but is also intentionally taken on due to time pressure and other reasons. For the success of a software project it is important to manage the amount of technical debt in order to minimize its negative impact. Similarly, risk management is an important part of software engineering. A risk is defined as some part of the software or circumstances of the development that imposes danger to the project's success. In order to minimize such dangers, the identification and handling of risk factors is crucial. The aim of this work is to determine the relation between technical debt and risk and whether these two issues can be managed jointly. For this purpose, a systematic literature review was performed. Overall, the relation between technical debt and risk management is barely covered in literature, but sources generally agree that the issues mutually affect each other, e.g., the accumulation of technical debt imposes several risks. Due to the similarity between the management techniques for both issues, they can be combined and united. Therefore, a joint management might be beneficial, but scientific evidence for this is missing.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*productivity, programming teams, software configuration management*

Keywords

technical debt, technical debt management, risk, risk management, software engineering

SWC Seminar 2018/19 RWTH Aachen University, Germany.

1. INTRODUCTION

The term of technical debt has gained progressively more importance since its coining in 1992. It draws a metaphor to the financial sector as it describes the postponement of difficult or time consuming tasks as a form of taking on debt which has to be repaid later. Without regular repayment of technical debt, it can accrue and lead to severe consequences, including but not limited to higher costs, decreased productivity of developers, lower maintainability and overall reduced project quality [13]. In order to minimize such negative effects, technical debt needs to be carefully managed, and various techniques for this purpose have been proposed. However, research in this field is still not encompassing and effective tools to manage technical debt are missing ([9], [12]).
This lack of adequate management procedures results often times in an increase of various risks, as for example the costs necessary to repay debts may be underestimated which can lead to missed deadlines or overall project failure. In contrast, it has also been pointed out that risks can increase due to technical debt that is not taken, as the implementation in accordance to coding standards is more difficult and time-consuming, such that timelines may not be met [14].
Both technical debt and risk are more and more frequently discussed in literature and various management techniques have been proposed. However, they have not been put into relation extensively. Mutual influences have been mentioned but not evaluated in detail. An assessment of the relationship might enable developers to manage both issues in a joint manner and thereby save valuable time and resources. This study aims to determine the relations mentioned in literature so far and thereby create a starting point for further investigation and joint management.
The rest of the paper is structured as follows: Section 2 will examine previous work on technical debt and risk in software engineering and give an overview about common management techniques of these separate issues. Section 3 will describe the procedure of this study on their relation and state concrete research questions. The results will be presented in section 4 and subsequently discussed in section 5. Lastly, section 6 will give a conclusion and some prospects on future work.

2. RELATED WORK AND BACKGROUND

In order to discuss the relation between technical debt and risk as well as a potential joint management of these issues, it is important to understand them separately at first. Therefore, this section will give an overview of their respec-

tive properties and common management techniques.
The term technical debt was coined in 1992 by Ward Cunningham [12] as he compared the usage of immature code to the financial concept of debt: The development can be accelerated by taking on debt, but it should be repaid as soon as possible, to minimize the interest. The repayment is conducted by rewriting the code, which becomes more and more complex the longer it is postponed and the more debt accrues. According to Cunningham, refactorings then require specialized programmers and engineering organizations can be damaged severely [8]. While this initial analogy seems easy to understand, there is still no single universally acknowledged definition today. After Cunningham's publication, the term became widely used in blogs, but only in 2010 it was taken up by scientific research. The Dagstuhl *Seminar on Managing Technical Debt in Software Engineering* in 2016 agreed upon two different viewpoints, concerning the discussion of design trade-offs with technical debt as a metaphor on the one hand, and the denotation of software artifacts indicating future costs on the other hand [12]. The seminar concluded with the following definition for technical debt:

> "In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability." [5]

This definition clarifies, that technical debt can not only occur in code, but is also related to documentation, architecture design and tests [5]. Accordingly, many subtypes of technical debt have been identified, such as architecture debt, build debt, code debt, defect debt, design debt, documentation debt, infrastructure debt and more [3].
The terms principal and interest were adopted in software engineering too, due to their direct relation to the concept of debt. The former refers to the benefit gained by taking on debt, i.e., savings of time and cost resulting from the imperfect code or design or, equivalently, the costs of completing the respective task properly ([5], [2]). Interest is the counterpart of this benefit, namely the cost resulting from the debt, which rises over time. Here, one can distinguish between recurring and accruing interest. Recurring interest is the consequence of decreased maintainability such as lower productivity, defects and poor quality. Accruing interest results from new software depending on the code containing technical debt, i.e., decreased evolvability [5]. Note that these consequences do not always emerge: If, for example, the affected module will not be used further so there is no need for maintenance and evolvement, the technical debt may not do any damage. This probability for negative impact is referred to as interest probability [2].
On average, 32% of the software development time in production systems is spent on repayment of technical debt, which offers an enormous potential of time saving when it is managed effectively [1]. By employing beneficial regularities, the incursion of technical debt can be avoided or minimized from the start. For example, a high frequency of small commits leads to higher complexity, while big changes tend to reduce technical debt and improve efficiency [3]. Techniques to manage technical debt after its incursion mainly involve the identification and monitoring of debt instances in order to repay them at a reasonable time during development [2]. This can be done by manually analyzing the source code in search for bad smells such as, e.g., god classes or code duplication, but such an evaluation is very costly and time consuming [13]. One of the most used tool to automatically identify instances of technical debt in literature is SonarQube. It works by scanning the source code for violations of predefined rules and charging a specified amount of time needed to repair any given violation. The accuracy of this tool has been shown to be similar to the results of manual evaluation by software specialists. However, the time estimated for the removal of technical debt items is not always appropriate, depending on the size and complexity of a project. While developers of a complex, sophisticated project may need twice as much time as recognized by SonarQube to fix a rule violation, the same problem might require only a fraction of the time in a smaller project. Therefore, researchers have concluded that this tool is not suitable to determine the amount of technical debt in a single project, but rather to compare its ratio between projects [9].
Continuous integration is another approach of managing technical debt. The method enables developers to discover errors and duplicate code earlier by having repeatable tests, which also supports the discovery of technical debt items. This validation becomes more difficult at the end of product construction, because functionalities and thereby the necessary test cases become more complex [3].
When identified accordingly, technical debt can also be continuously repaid by continuous refactoring. This is a good way to prevent its accumulation to a critical level and also supports managers in the communication with developers, such that old solutions can be reused and optimized more easily. [3].
CAST is another approach, which calculates technical debt based on coding violations associated with the quality attributes security, performance, robustness, transferability and changeability. The violations are classified into groups of low (10%), medium (25%) and severe (50%) degree of violation, whereby, depending on the category, violations do not always have to be remedied [7].
Additionally, technical debt can be investigated with various analysis tools, for example to identify its subtypes such as architectural debt. These analysis tools use, e.g., standardized code metrics or logical flow charts. A requirements analysis of the current status can give an indication to which level of severeness a debt item has progressed. By documenting the progress of software development, an estimate of the technical debt can be given based on the detailed description of the procedure for developing the software. Another approach is a bottom up enumeration of technical debt, starting with trivial mistakes and progressing to rather unobtrusive ones. The problem here is, however, this requires maintenance from the beginning, otherwise the generation of this overview is too expensive and time-consuming [10].
In addition to technical debt, various risks should be managed in software engineering in order to prevent, e.g., rework and additional costs. Risk management is defined as a process of detection, analyzing and dealing of risk factors in a project. Risk factors are, for example, unrealistic time planning, lack of budget and insufficient skills. All of these factors can contribute to software being improperly devel-

oped [1]. A good management, however, can enhance the development procedure and thereby the quality of software [7].

There are many different approaches to manage risk. One of them is the use of checklists which are created from experienced stakeholders and project managers in related software projects. These lists contain important risk factors identified in similar projects, which can be found more quickly when developing new software, e.g., personnel shortfalls or late changes of requirements [4]. Comparing such lists with the circumstances of a given project is a quick and low cost opportunity of managing risks in software engineering. However, the use of checklists entails difficulties, too. Many different versions exist, but it is not clear which one of them is the best. Furthermore, there are so many potential risks in a software project that identifying and managing them by checklists is not effectively possible, especially concerning large projects [6].

Another approach of managing risks is the use of non-process based analytical frameworks. The multitude of risk factors are grouped into categories so that they can be managed and treated as a group. Various types of grouping exist in the literature. For example, one grouping option introduced by Cule et al. includes the categories client, self, task, and environment, referring to the source of risks. Another option was introduced by Lam in 2014, where the categorization is life cycle based, meaning the risk factors are categorized based on the phase of software development they emerged in. Putting risk factors together in groups increases treatment efficiency considerably and minimizes the effort involved. Furthermore, the categorization in groups supports the discovery of further problems or risks. But as with checklists, it is not clear which tool or grouping is best. A previously used or created tool is not always sufficient for other projects and in some cases a combination of multiple tools might be beneficial [6].

Process models are another approach that is presented in the literature and has the most use in practice. Communication between developers participating in the software is an important aspect of any jointly cultivated software project, since changing one component can have a huge impact on other parts of the software. In order to recognize the extent of such impacts, a number of steps must be performed. First of all, the respective causes and possible consequences need to be recognized and identified in detail. Then, the consequences are analyzed in order to determine resulting risks and thirdly, these risks are evaluated such that remediation activities for the most severe risks can be prioritized. After this series of steps of identification, analysis, and evaluation, risks can be monitored, and documented and treated appropriately [11]. The division of this procedure into individual steps facilitates the application of corresponding tools, which can simplify each step. The implementation of risk management with the help of process models should be carried out iteratively during development, such that the emerging risk factors are recognized early and environmental changes are integrated [6].

The approaches presented here can also be combined and used together, as they are partially incomplete and complement each other. E.g., process models do not offer concrete solutions. Through the combination of multiple tool options, risk management can be completed successfully and efficiently [11].

To conclude this section, the management techniques for technical debt and risk in software engineering share significant similarities. For both issues, identification and monitoring are the main activities. By expanding the criteria during the identification process, it may be possible to handle debt and risk at once. A significant difference is the considered scope: While technical debt relates only to software artifacts, i.e., code, documentation or architecture design, risk factors can be found in external circumstances of the project as well, e.g., among stakeholders or in characteristics of developers. It has not been discussed in detail, whether a joint management of both issues is beneficial. In order to avoid unwanted side effects, the ways in which technical debt and risk can influence each other should be identified at first. This gap in research so far is dealt with in the following sections.

3. METHOD

The aim of this study is to determine the relation between technical debt and risk management. Therefor, we roughly followed the systematic literature review methodology proposed by Barbara Kitchenham [10], but omitted or shortened some parts like the Study Quality Assessment due to the limited scope of the seminar. In the following, our procedure is depicted in detail.

3.1 Research Questions

For a precise literature search we formulated the following research questions based on the gap discovered in related work:

RQ1: What is the relation between technical debt and risk?
Both technical debt and risk are common issues in software development, but their relation is not entirely clear. This research question is intended to determine how they are connected and to what extend one of them causes or mitigates the other.

RQ2: Can technical debt and risks be managed jointly and if so, how?
For a successful project, both technical debt and risk should be managed effectively. Depending on their relation, it might be possible to manage them jointly by making use of that relation.

3.2 Search Strategy

In the initial search, the focus was on narrowing the number of results by filtering out irrelevant papers, e.g., those related to financial debt. Furthermore, we aimed to disregard papers dealing exclusively with technical debt or risk, since we were looking specifically for connections between both of these issues. After considering various combinations of "technical debt" and "risk management", the following search query was selected:

- "technical debt" in all metadata
- "risk" in full text and metadata
- "management" in full text and metadata

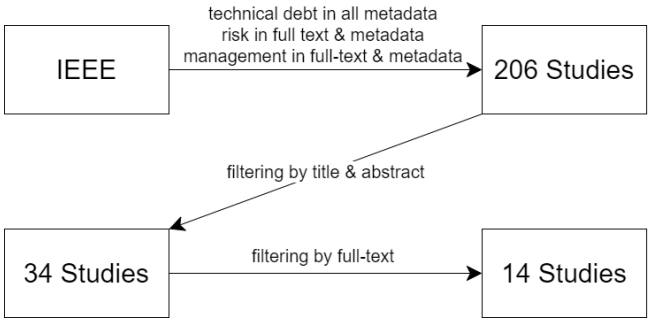

Figure 1: The procedure and intermediate results of the literature search

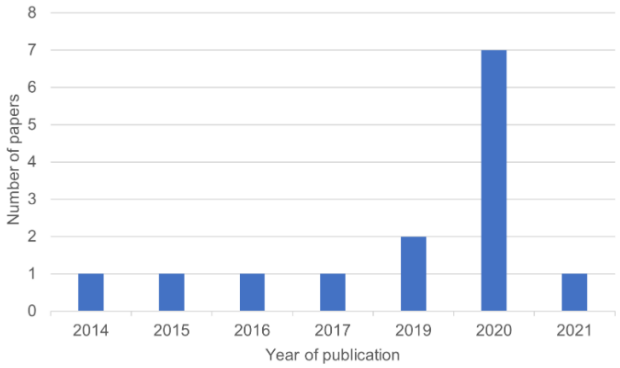

Figure 2: The publication years of the papers selected for this mapping study

3.3 Study Selection

Due to the large number of results obtained in various databases, the limitation to a single literature database was estimated to give a sufficient overview of the topic. For this study, IEEE was used, but we expect other common databases to yield similar results. In IEEE, 206 Conferences and Journals were obtained, which were managed in Citavi. These were portioned evenly between the two authors to filter them based on their title and abstract: The main inclusion criterion was the mentioning of technical debt in the context of software projects as well as some notion of risks or the probability of future impairments. After this first filter, 34 studies remained. The second filtering was based on the full text of the studies, where each author filtered the studies selected by the other one in the previous step. Here, only studies which inferred some kind of relation between technical debt and risk were selected. An explicit denial of any relation would have been included as well but was not found in any study. In the end, 14 studies remained. The complete procedure of study selection is illustrated in Figure 1.

3.4 Data Extraction

After the study selection, each author analysed half of the remaining studies to determine the inferred connection of technical debt and risk. It should be noted, that no study was found which focuses on the relationship between these issues, except one which is strictly limited to security debt and security risk ([9]). General connections were drawn only in minor subsections or as side notes. Such mentions were extracted individually for each study and summarized with respect to the specific context within the paper.

3.5 Data Synthesis

Due to repetitions between studies, the extracted data was summarized at first. Some explanatory details about the studies' context and management techniques were disregarded or complemented in Section 2, as they are not directly concerned with the topic but necessary for reference. Then the data was grouped into related concepts in order to create a comprehensible structure and emphasize the variety of connections between technical debt and risk.

4. MAPPING RESULTS

The relation between technical debt and risk has not been examined in detail so far, but several researchers have drawn a connection in recent years. Figure 1 displays the distribution of papers mentioning some relation, that were found in this study. The earliest mention was published in 2014 and the topic was addressed a maximum of seven times in 2020. While this might indicate a significant increase of interest in the topic, the filtering only resulted in one paper from 2021 and did not find any relevant paper published this year.
The relations between technical debt and risk discovered in literature are manifold and will be described in the following subsections.

4.1 Risks resulting from taking on debt

Various project risks are significantly effected by technical debt, but the risk of high effort and duration is the main effect of technical debt on project level [1]. This leads, e.g., many startups into bankruptcy because they fail to manage technical debt in the early stages, such that it accumulates and causes severe defects and costs [3]. Without management, technical debt can lead to the degeneration of a system's architecture [2].
Due to these potential consequences, the emergence of technical debt itself can be seen as a risk. Tan et al. investigated the co-occurring of multiple instances of technical debt in Python to discover whether the presence of some types of technical debt can be used to estimate the risk of additional technical debt to emerge. According to their results, most associations between technical debt items are random, but some types have indeed a high chance of occurring simultaneously. For example, a high cyclomatic complexity tends to increase the cognitive complexity of code. Furthermore, the amount of a specific subtype of technical debt called defect debt increases, if the software is not tested completely, and overall the accumulation of bad smells might increase the risk of specific design problems to emerge soon. Another finding of the study is that co-occurrence mostly happens very quickly and is then hard to eliminate [13].
Rindell and Holvitie connected technical debt to security risk, i.e., they found that for example the data integrity of the resulting software can be impaired by specific items of technical debt. They refer to such items as security debt and argue that previously unknown debt can be identified by means of a security evaluation, as they are discovered as the source of security risk. This is an unusual approach and can not be easily transferred to other types of technical debt or risk [9].

4.2 Risks resulting from repaying debt

In addition to the accumulation of technical debt, its management and repayment can also impose risks. E.g., the approach of continuous integration for discovery of technical debt items can impose the risk of a shifted focus away from the actual functionalities of the software: Because the tests become more and more complicated towards the end of production and need to be updated continuously, developers may spend too much time on these tests and fail to finish the implementation of software functionalities in time [3].
A crucial method for the repayment of technical debt is the refactoring of impacted code. In fact, technical debt is a main motivator for refactorings in software projects, since high complexity and low readability of code can make bug fixing very difficult even for experienced developers. Such rewritings or reorganisations of code are a common way to improve its quality and comprehensability. However, refactoring can also impose risks: According to a survey with 328 software engineers, 50% of developers fear that refactorings will introduce new bugs and other side-effects. Especially refactorings in relation to hierarchies often induce faults. Those related to the relocation of code are less likely to do so [8].

4.3 Risk as a decision factor in technical debt management

Martin Fowler created a way to classify technical debt items based on the way risks were considered during the decision process to take on the debt with his Technical Debt Quadrant. He classified technical debt items into the categories reckless deliberate, reckless inadvertent, prudent deliberate, and reckless inadvertent, which correlate to the following attitudes towards risk [14]:

reckless deliberate	The risks resulting from technical debt are ignored and the resulting consequences were not taken into account.
reckless inadvertent	The resulting risks and consequences are not apparent during development.
prudent deliberate	The resulting risks and consequences could be identified and analyzed during development.
reckless inadvertent	The risks and consequences would be analyzed if they were evident.

The decision whether or not to take on technical debt should always be made in consideration of both the advantages and risks. While the former are usually imminent, e.g., reduction of time pressure, future risks are often unknown and could affect various aspects like support costs and code complexity [15]. The knowledge of commonly co-occurring types of technical debt may help developers to foresee possible consequences of taking on debt [13] and therefore to avoid additional project risks.
The following attributes have been found to be key factors for the decision on whether or not or at what time to repay technical debt: Severity, existence of a workaround, urgency of fix required by customer, effort to implement the fix, risk of the proposed fix, and scope of testing required. Usually, it is beneficial to repay debt rather sooner than later, as the interest increases as long as the repayment is delayed. Especially in large systems refactoring becomes harder when it is delayed, which also increases the risk of not meeting timelines. However, there have been instances of technical debt that did not incur any interest, even when their repayment was delayed, so they do not impose any risks [5]. In any case, risk is a fundamental component of the subject of technical debt and when it needs to be repaid [11].

4.4 Prioritization of technical debt based on risk

Since the resources for debt management are usually limited, it is important to prioritize the repayment of those debt items which impose the greatest risk over less dangerous ones. Codabux and Williams have developed a framework for this purpose, to categorize technical debt items into low, medium and high risk items. Thereby, the items are rated through the Analytical Hierarchy Process, which is a comparison technique that can be conducted with various metrics. In this case, suitable metrics would be, e.g., the impact of the debt item estimated by developers, its location in source code or its relationship to crucial system functions [2].

4.5 Identifying risks based on technical debt

While most of the literature addresses risk as something to be considered during technical debt management, the same holds vice versa, as the technical debt present in a software project can be utilized to manage risks. Through the categorization of debt items into different subtypes, possible risks in the individual development phases of the software can be identified at an early stage and remedied, depending on the damage [9].

5. DISCUSSION

The mapping results verify our expectation that technical debt and risk are connected, but the data situation is insufficient. Still, in the following the research questions from Section 3.1 will be discussed based on our results.

RQ1: What is the relation between technical debt and risk?
The relationship between technical debt and risk management can be summarized as follows. Technical debt can be seen as a risk factor in software development due to the uncertainty of the emergence and severity of its negative consequences. In a way, increasing risk is an intrinsic property of technical debt. A decisive difference to other risk factors is that technical debt is often taken on, intentionally or subconsciously, in order to circumvent current shortages or problems.
Irrespective of whether technical debt is considered as a risk factor itself, it can lead to the emergence or increase of various other risks, such as project failure, additional development cost, timeline transgressions and many more. These consequences and the probability of their occurrence vary between different items of technical debt, depending on the kind of debt, its severeness, repayment and the further development of the respective artifact.
Furthermore, the management and repayment of technical debt items bears risks. This might seem contradictory, be-

cause the management is supposed to relieve the risks of technical debt, but the procedure can be very resource intensive and thus hinder actual product development. Besides, refactorings with the purpose of debt repayment are also prone to error and may incur additional or even more risky debt.
Despite these connections found in literature, there has not been any explicit comparison of the amount of risk and technical debt in software projects yet, to the best of our knowledge. A statistic of, e.g., the number of project failures or releases behind schedule in dependence of the incurred technical debt could give a more definite and detailed indication of the correlation between technical debt and risk.

RQ2: Can technical debt and risks be managed jointly and if so, how?
When dealing with technical debt and risk management, the same or very similar procedures can be identified. In both domains, the remediation of issues is less costly, financially and in terms of time, when they are identified and managed earlier. This does not mean, however, that risk management strategies can be directly applied to technical debt or vice versa. Both issues occur in various different forms with different outcomes depending on the individual software project, such that a tailored combination of multiple management strategies is usually advisable.
We expect the joint consideration of both issues to be beneficial, because an effective management of risks can support the management of technical debt. With knowledge of the risks present in a software project, the repayment of those debt items which cause the greatest risk can be prioritized. Such repayments can also be timed adequately in dependence of further development and usage of respective artifacts. Additionally, during the decision whether or not to take on technical debt, the risks stemming from potential shortcuts should be contrasted with those that the debt item is intended to avert. This can help developers to determine whether it is actually profitable to go into debt in each individual situation.
Note that the joint consideration as described here is only a suggestion based on the connections between technical debt and risk found in literature. It has not been put to the test and needs further investigation to determine its effectiveness in practice.

5.1 Implications

These results suggest that practitioners in software engineering should consider risks during the management of technical debt and vice versa. Since the management techniques are similar, they can be combined and extended to both issues, such that time and resources can be saved. For every decision on taking on debt, the risks resulting from this specific debt item should be carefully contrasted with those it is intended to reduce.
Similarly, researchers should consider the risks coming from technical debt and its repayment as well as the probability for technical debt in dependence of various risk factors during their work. Knowing about such connections enables them to draw more specific conclusions and determine the reasons for project failures or other problems. Besides, the benefit of a joint management as suggested in this paper should be statistically verified.

5.2 Threats to validity

In the following subsections, two possible threats to the validity of this study will be discussed. Thereby, internal and external validity is separated.

5.2.1 Internal validity

A threat to the internal validity of this study may be the limitation of the literature search to only IEEE. The picture of the relation between technical debt and risk management drawn in this study might be incomplete, because only one database was taken into account. However, IEEE is known to be encompassing in the field of software development and an initial rough assessment of other databases such as ACMDL, ScienceDirect and Scopus revealed similar results.

5.2.2 External validity

Concerning external validity, one should note that this study deals with the concepts of technical debt and risk as a whole, respectively. Especially technical debt can be categorized in many different subtypes such as architecture debt, code debt or defect debt. Each of these types as well as their repayments may have individual implications on risk, which is not considered in this study. Such differences might impact the degree to which it is possible to manage the respective debt item in a joint manner with risk. In fact, even the same type of debt can have different consequences in different software projects, as project properties and development processes differ. Therefore, a joint management technique that worked well in one project might not be transferable to other ones.

6. CONCLUSION

All in all, similarities between managing technical debt and risk can be noted: For both issues, the detection of relevant instances constitutes a big part of the management techniques. Thereupon, the severity of found instances needs to be assessed and monitored in both cases, such that their remediation can be prioritized and planned effectively. Performing these procedures jointly for technical debt and risk might therefore be a possibility to save time and other resources. This is not entirely possible since technical debt is limited to project artifacts such as code and architecture design, while risk factors can come from external circumstances like developer experience or stakeholder involvement. Nevertheless, such circumstances can also influence the occurrence of technical debt, so they should be regarded in debt management too.
Risk should play a significant role in the assessment of technical debt items, because the incursion as well as the repayment of technical debt items can impose risks. The same holds vice versa, since the accumulation of technical debt and especially the growth of its interest can be considered as a risk with varying severity.
Note that these statements are mainly speculative so far and are not sufficiently backed with real data. The relation between technical debt and risk and especially the effectiveness of a joint management of these issues requires considerably more research. E.g., as mentioned before, a large scale study on the frequency of project failure in dependence of technical debt could give a better indication of the correlation between technical debt and risk. Such a statistic might also consider and compare projects where technical debt and risk

are already managed jointly to some degree. Additionally, developers can be questioned in a survey on whether or how often they have experienced a project risk resulting from technical debt or vice versa. The questionnaire should also inquire information on how the connection was identified, how such situations were handled and which steps were initiated to avoid their repetition. Since developers may not be able to answer these questions precisely, it might also be beneficial to accompany entire software projects in order to scientifically monitor any occurrence of technical debt or risk as well as their management.

References

[1] T. Addison and S. Vallabh. Controlling software project risks: an empirical study of methods used by experienced project managers. In *Proceedings of the 2002 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, pages 128–140. Citeseer, 2002.

[2] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, 70:100–121, 2016.

[3] N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola. Towards an ontology of terms on technical debt. In *2014 Sixth International Workshop on Managing Technical Debt*, pages 1–7, 2014.

[4] T. Arnuphaptrairong. Top ten lists of software project risks: Evidence from the literature survey. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, volume 1, pages 1–6, 2011.

[5] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman. Managing technical debt in software engineering (dagstuhl seminar 16162). In *Dagstuhl Reports*, volume 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[6] P. L. Bannerman. Risk and risk management in software projects: A reassessment. *Journal of systems and software*, 81(12):2118–2133, 2008.

[7] B. Boehm. Software risk management: principles and practices. *IEEE Software*, 8(1):32–41, 1991.

[8] W. Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1992.

[9] J. S. de Jesus and A. C. de Melo. Technical debt and the software project characteristics. a repository-based exploratory analysis. In *2017 IEEE 19th Conference on Business Informatics (CBI)*, volume 1, pages 444–453. IEEE, 2017.

[10] B. Kitchenham. *Kitchenham, B.: Guidelines for performing Systematic Literature Reviews in software engineering. EBSE Technical Report EBSE-2007-01.* 01 2007.

[11] J. Masso, F. J. Pino, C. Pardo, F. García, and M. Piattini. Risk management in the software life cycle: A systematic literature review. *Computer standards & interfaces*, 71:103431, 2020.

[12] F. Ocker, M. Seitz, M. Oligschläger, M. Zou, and B. Vogel-Heuser. Increasing awareness for potential technical debt in the engineering of production systems. In *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, volume 1, pages 478–484. IEEE, 2019.

[13] L. Rantala. Towards better technical debt detection with nlp and machine learning methods. in 2020 ieee/acm 42nd international conference on software engineering: Companion proceedings (icse-companion)(pp. 242–245), 2020.

[14] E. Tom, A. Aurum, and R. Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516, 2013.

[15] R. Zablah and C. Murphy. Restructuring and refinancing technical debt. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pages 77–80, 2015.

APPENDIX

SELECTED STUDIES

[S1] S. Biffl, F. Ekaputra, A. Lüder, J. Pauly, F. Rinker, L. Waltersdorfer, and D. Winkler. Technical debt analysis in parallel multi-disciplinary systems engineering. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 342–346. IEEE, 2019.

[S2] Z. Codabux and B. J. Williams. Technical debt prioritization using predictive analytics. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 704–706. IEEE, 2016.

[S3] R. Colomo-Palacios et al. Continuous practices and technical debt: a systematic literature review. In *2020 20th International Conference on Computational Science and Its Applications (ICCSA)*, pages 40–44. IEEE, 2020.

[S4] M. de León-Sigg, S. Vázquez-Reyes, and D. Rodríguez-Ávila. Towards the use of a framework to make technical debt visible. In *2020 8th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pages 86–92, 2020.

[S5] Z. Li, Q. Yu, P. Liang, R. Mo, and C. Yang. Interest of defect technical debt: An exploratory study on apache projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 629–639. IEEE, 2020.

[S6] A. Martini and J. Bosch. Towards prioritizing architecture technical debt: Information needs of architects and product owners. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, pages 422–429, 2015.

[S7] A. Mayr, R. Plösch, and C. Körner. A benchmarking-based model for technical debt calculation. In *2014 14th International Conference on Quality Software*, pages 305–314, 2014.

[S8] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia. An exploratory study on the relationship between changes and refactoring. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 176–185. IEEE, 2017.

[S9] K. Rindell and J. Holvitie. Security risk assessment and management as technical debt. In *2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, pages 1–8, 2019.

[S10] L. A. Rosser and J. H. Norton. A systems perspective on technical debt. In *2021 IEEE Aerospace Conference (50100)*, pages 1–10, 2021.

[S11] S. Soares de Toledo, A. Martini, A. Przybyszewska, and D. I. Sjøberg. Architectural technical debt in microservices: A case study in a large company. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 78–87, 2019.

[S12] M. G. Stochel, P. Chołda, and M. R. Wawrowski. On coherence in technical debt research: Awareness of the risks stemming from the metaphorical origin and relevant remediation strategies. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 367–375. IEEE, 2020.

[S13] J. Tan, D. Feitosa, and P. Avgeriou. Investigating the relationship between co-occurring technical debt in python. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 487–494. IEEE, 2020.

[S14] A. Zalewski. Risk appetite in architectural decision-making. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 149–152, 2017.

Exploring Technical Debt Management in Project-Based and Product-Based Software Development

Simon Hessel
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
simon.hessel@rwth-aachen.de

Tobias Raaf
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
tobias.raaf@rwth-aachen.de

ABSTRACT

Technical debt management (TDM) has been accepted as an essential part of today's software development process. As up to 75% of software development effort is spent on maintenance, managing technical debt becomes a key factor for successful software development. Until now, the research on TDM was focusing on company, project and tool-specific case studies, but the impact of different decisions concerning the development process and approach has not been brought into sharp focus. This paper concentrates on understanding the influence of different software development approaches (i.e. product-based and project-based software development) on TDM strategies. We believe that there are some dependencies between the development environment/conditions and TDM.

To find practical evidence in this context, we conducted a systematic literature review of existing primary studies (e.g. case studies and developer surveys) on TD. We will compare TD concepts as well as individual steps taken to manage TD. Our results show that certain TD concepts differ strongly between development modes, e.g. decreased TD visibility in project-based software development or budget-shortages as a predominant TD cause in projects which call for proposals. While multiple other TD causes, as changing requirements or release pressure, are common for both development approaches, they often differ in detail. There are also differences in TDM as, for example, project-based teams tend to implement the action of TD measurement more often and they have a higher necessity of implementing TD documentation. On the other hand, TD repayment differs strongly depending on the exact circumstances of the project/product, and not only depending on the development mode. For TD communication, we made assumptions that project-based teams could have a higher focus on this action, but this could neither be proven nor disproven. Other TD concepts and TDM actions differ in detail rather than in their general implementation. Lastly this study shows the absence of studies solely focusing on the development mode, therefore further research needs to be done.

SWC Seminar 2022 RWTH Aachen University, Germany.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*productivity, programming teams, software configuration management*

Keywords

Technical debt, technical debt management, product-based software development, project-based software development, software development, systematic literature review

1. INTRODUCTION

Technical debt (TD) first used by Ward Cunningham[3] in 1992 is a metaphor to describe decisions and practices in a software development team that solve a current problem fast and/or easy by neglecting code and architectural design. Similar to its financial counterpart[1], TD can therefore accumulate interest in the form of increased cost and time to be spent in later phases of the project.

Managing said debt also called **TD management** (TDM) describes the practices of monitoring, quantifying, reducing and handling TD in a software development environment[7]. With ongoing time, TDM was categorized into activities, for a clean separation between different approaches and practices[7]. TD repayment, TD identification and TD measurement are the most commonly discussed activities, these terms will be explained in detail in section 2 of this paper.

This paper will focus on comparing TD concepts and TDM in **project-based** vs **product-based** software development environments. Where project-based environments have a single customer and a fixed time frame, which should be complied to[5], product-based environments have a flexible scope and target an open market with multiple possible, but yet unknown customers. Because the maintenance of ongoing software development projects is estimated to take up 50-75% time spent[1] TD and TDM is an important research field. In the current scientific literature, there are many case studies on the topic of TD and TDM, in which a specific tool or method is being applied to a small group or company or multiple developers are surveyed on TD and TDM. Additionally, there are multiple systematic mapping studies like one performed by Zengyang Li et al. named "A systematic mapping study on technical debt and its management"[7], which visualizes and combines the state of scientific research on TD and TDM until 2014. However, there has been no

research conducted on the effectiveness of different TDM activities when comparing product- and project-based development environments. In this paper, we want to focus on this gap, give insights into certain studies and answer the following research questions on the topic of TDM:

RQ1 How do TD concepts differ between product- and project-based development?

RQ2 How does TDM change in product- vs project-based development?

To answer these questions we will do a systematic literature review (SLR)[2], in which we will classify the selected papers and their studied software development teams, see more in section 3, into project- or product-based development environments and analyze the applied TDM strategies as well as impact on TD concepts.

This paper is organized as follows: Section 2 explains the underlying concepts of TD, TDM and project- vs product-based. Section 3 describes the limits of this paper and how the systematic literature review was performed. Section 4 concludes the results received by the SLR. The discussion of said results is performed in section 5, and section 6 contains the conclusion reached by this study.

2. BACKGROUND

In this chapter, we will go into detail on the most important concepts used in this paper.

In Ward Cunninghams definition of TD the focus was lying on "code debt" i.e. compromises to code quality in favor of faster releases[3]. Over the years, the definition was broadened to include other parts of software development, which then became their own TD types[7]. TD being a financial metaphor evolved to being used as communication ground for non-technical and technical people, which has a high possibility to backfire as not every financial term and concept attached can be applied and translated to its technical counterpart[1]. Therefore, in the following chapters we will use the following definition of TD: Decisions in a software development environment that are made intentionally or unintentionally to speed up or simplify the current task, therefore causing more work in the future.

To handle the consequences of these decisions, TDM is implemented. TDM stands for a group of activities/approaches to handle TD in software development environment and therefore eliminate unintentional TD[7]. TDM comes in form of tools and concepts to be used in software development. Most tools e.g. (IDEs, separate tools) are using the respective source code as input to make use of static analysis, thereby mostly focusing on code TD. TDM concepts are often coupled with agile development practices like sprint retrospective and reviews, but also other concepts like test-driven development (TDD) can be part of TDM[S1].

TDM actions can be categorized into certain steps which are listed and explained below:

- TD repayment: Removing accumulated TD, e.g. code refactoring[7].
- TD identification: Finding/Tracking planned or unwanted TD, e.g. source code analysis tools [7].
- TD measurement: Calculating positive and negative impacts of TD[7].
- TD prioritization: Ordering TD by relevance of repayment[7].
- TD communication: Provide transparency of the TD state to non-technical team members[7].
- TD prevention: Reducing TD before it is created[7].
- TD representation/documentation: Create visualizations of the state of TD[7].
- TD monitoring: Tracks the TD changes progressively over time[7].

For improved readability, we graphically emphasized the names of the TD concepts and TDM activities in the result chapter of this paper.

For related work, we would like to highlight Zengyang et al. who conducted a systematic mapping study on TD and TDM, which is concerned with highlighting the current scientific status of TD TDM[7]. We identified a research gap as no prior research has been conducted on the implications of different TDM strategies depending on the development environment.

Depending on the project mode, different risks are present, other restrictions to the cost and project time frame apply and members with varying technical knowledge are part of the team on client and vendor site. Two main development environments, and the two we will focus on in the scope of this paper, are product-based and project-based development.

Jez Humble defines "product vs project" by classifying projects to "have an end-date and a single customer, and we care about scope, cost and hitting the date"[4]. Products, on the other hand, "are evolving continuously, we have multiple customers, and we care about a broader set of risks"[5]. According to this definition, the three most important attributes, to distinguish the development environment by, are: The presence of a project time frame, the presence of a single, responsible customer, and whether multiple teams are working together on the product. The first two attributes are derived directly from the definition. The third attribute aims to reflect the evolution of the product and emphasizes the existence of self-contained project terms. Any other means of measuring the evolution of a product with the amount of information given per reviewed study is impossible or out of scope for this paper. This definition will be the foundation of the mapping of all reviewed studies to the corresponding development environment.

3. METHODOLOGY

After laying the foundations with the theoretical background, this chapter will elaborate on the methodology. First, the limits to our work are defined and the SLR as a method is explained in general. Afterwards, the selection process and data synthesis in this paper is explained in detail.

3.1 Limits to our work

As this paper is part of the university seminar, the scope of the paper had to be limited. Therefore, for the systematic literature review (SLR) we only included 3 scientific

databases (ACM, Scopus, IEEE) and are not using the snowballing principal[6] when selecting the papers to be used in the SLR.

3.2 Systematic literature review

A SLR was selected as the paper type to fill the current research gap of TDM in relation to product- vs project-based development environments. This method was used to structure and document the evaluation of the empirical data conducted by prior studies. Since this paper was conducted as part of a university seminar, there was no option of peer reviewing the review protocol before finalizing the paper. All other guideline steps were performant as proposed by Kitchenham et al.[2]

3.2.1 Search strategy

The first step taken to ensure that only relevant papers are included and reviewed was setting the correct search criteria. To exclude all papers in our search scope that do not address TDM, the main search string was set to *"technical debt management"*. By using each database's query engine, the results were filtered on top to only include one of the following strings in its title *"project"* or *"product"* or to include *"project-based"*, *"project management"*, *"product-based"* or *"product management"*. The concrete queries can be found in section A. Moreover, only peer-reviewed research was included. The resulting papers were published between 2011 and 2021. These can be seen in table 1.

Database	Amount	Without duplicates
ACM	29	29
Scopus	22	17
IEEE	22	13
Total		65

Table 1: Database search results

Afterwards the following exclusion and inclusion criteria were applied by going over the abstract and conclusion of each paper first and reading the whole paper second.

Exclusion

- The paper is a duplicate from another database
- The paper has no publicly downloadable version
- The paper was not written in English
- The paper focuses on a project with too small scope e.g. is looking specifically at the implementation of a feature and not a more general project flow

Inclusion

- The paper discusses TDM strategies and their impact on the development environment
- The paper is peer reviewed, a case study or survey

Taking into account the exclusion and inclusion criteria twelve papers were left as part of the SLR, they were distributed among the databases as follows.

The work of reading through the papers and applying the criteria to select them as study material was divided evenly among the two authors. Furthermore, both researchers working on this, were not working on a scientific study in the field of TDM before and had their first deeper insights into this topic while preparing the study and this paper.

Database	Amount
ACM	3
Scopus	5
IEEE	4
Total	12

Table 2: Database filter results

3.2.2 Mapping strategy

Although some papers discussed in this systematic literature review, most notably the case studies, do describe the observed scenario and development environment, most do not explicitly state if the development environment is project-based or product-based. As this paper aims to compare results of studies depending on the development environment, we will need a foundation to map all studies to a development environment, when possible. Therefore, we derived a list of questions or criteria from the definition and the list of central attributes of the development environment stated in section 2:

- Is the development environment explicitly stated?
- Does the described development process have a time frame?
- Is the team working for/with one customer, or is it delivering a product to multiple, possible customers?
- Is one team accompanying the whole product development lifecycle? Or are different teams working on the product in different product development phases, e.g. maintenance after product release?

The first criterion takes into account whether a paper defines its observed development environment explicitly. If this criterion is clearly met, the definition by the paper itself is used for the mapping and the remaining criteria are omitted for this paper. The second criterion reflects the first central attribute of a project-based development environment, a positive response to this question implies a project-based development environment. Similarly, the third criterion reflects the second central attribute of a project-based development environment by questioning whether the presence of a single customer working directly with the development teams is stated, the presence of a single customer indicating a project-based development environment. The last, more complex, attribute of product-based development environments is covered by the last criterion. In this case a single team working on the whole product implies a product-based development environment.

Because of the major management differences between product- and project-based development, the base assumptions of when and how TD is acceptable are different. In the next sections 4 and 5 we want to highlight specifically how the different TDM actions may be prioritized differently depending on the project-mode. Therefore, in our next step the results were manually categorized into project-based, product-based, mixed and unknown development environments using the questions outlined in this section.

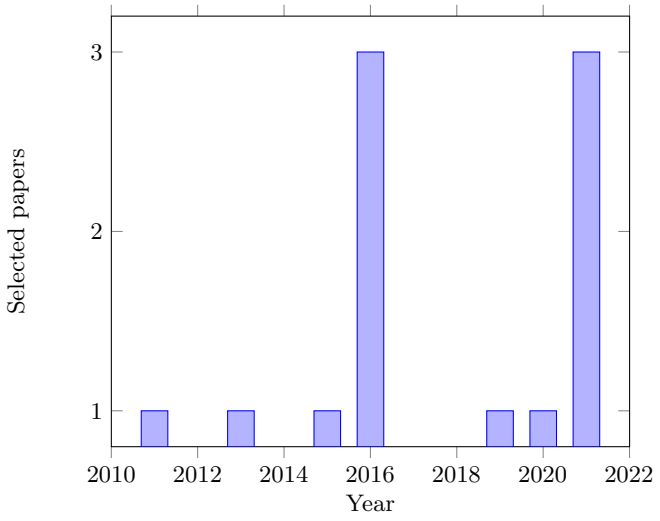

Figure 1: Year allocation of primary studies

3.2.3 Mapping by development environment

The process of mapping the studies was performed by mainly focusing on the description of the survey participants or case study context, if it was a study that conducted a survey or a case study respectively. The questions were answered separately with the information provided in this description. In some cases such a description was not given and there was no information found on the context otherwise. Those studies were mapped to the "unknown" category. While reviewing the answers, no contradictory statements were found. Therefore, no weighting of the statements among each other had to be carried out. The categorization of each study was then derived from the summary of the statements. The resulting distribution of the papers among the different development environments can be found in table 3. The answer for each mapping question per paper can be found in section C.

	Project-b.	Product-b.	Mixed	Unknown
Studies	[S3] [S7] [S11]	[S4] [S6] [S10]	[S1] [S2] [S8] [S9] [S12]	[S5]
Total	3	3	5	1

Table 3: Paper categorization by development environment

4. RESULTS

When comparing the different case studies and surveys, categorized by their development environment, it becomes obvious that there are many similarities between product- and project-based development processes. For example, TD categories are often distinguished along the same boundaries or main groups[S1] and the most integral steps to TDM are part of product-based, as well as, product-based development[S12]. But there are some differences in TD concepts and TDM depending on the development environment. In this result chapter, we will first focus on differences in TD concepts, as these can lead to a higher necessity of certain TDM steps and in this way differences are transferred to TDM, and afterwards we will focus on TDM steps which differ between the development environments.

4.1 Differences in TD concepts

The common TD concept most papers focus on is the cause of TD. While skill shortage in the developer team is a generally present cause of TD for both approaches[S6], Some of the main causes are frequently discussed for both development environments, but differ in detail. Deadlines as a TD cause, for example, are discussed in project-based and product-based development. While for project-based development the customer, pressuring for an early implementation of a feature or visible changes[S7], and the budget are main reasons for deadlines[S2], for product-based development it is mostly the need to deliver new features to the targeted market, which generates pressure[S6].

Another similar case for both product- and project-based development, which solely differs in detail are changing requirements. For product-based development, this often means a changed product vision, which is necessary to reach a broader group of possible customers[S6]. For project-based development, this can occur as well, whenever the customer's requirements change[S2]. While the underlying reason for the cause is different, the effect stays the same, the TD is generated.

There are some TD causes however, which only occur in either product-based development or project-based development. An example for this is fixed budgets in competitive projects which call for proposals. When development teams make such a proposal, they often have to include an estimated budget without being able to negotiate requirements with the customer beforehand. This motivates shortcuts to speed up feature implementations at the expense of code quality[S2]. On the other hand, there is a tendency in project-based development to require a more detailed architectural design, which is made in consultation with the customer before even starting the coding process, while product-based development tends to a more spontaneous design approach[S2]. This can lead to inappropriate planning, which is a cause of TD[S8].

TD consequence is an attribute which is mostly similar between the two approaches, with the most extreme consequences being unsolvable bugs/defects or even product or project termination[S1]. Another relevant TD concept is visibility. For TD visibility, project-based development has a higher risk of needing extra measures to assure a high visibility of TD because whenever a project ends and a follow-up project is started with a new team, intrinsic knowledge of each former project member is lost[S2][S6]. A necessary countermeasure to this may be a greater focus on TD communication and TD representation/documentation in TDM.

While these differences in TD concepts may affect TDM, there are certain characteristics of the two development environments, which directly influence TDM.

4.2 Differences in TDM

Same as for TD concepts, TDM is largely similar, with differences in detail. The goal of TDM, handling TD in a way to balance interest and repayments, stays the same. When discussing the individual TDM steps, while all steps are applied for both development environments[S12][S5], some gain greater focus as others. Broken down for each TDM step, this yields different insights:

While **TD identification** is arguably the most costly step of all[S3], no major differences are noticeable. However, one reason why TD seems hard to identify and categorize is that TD propagates throughout the whole development process[S5], meaning architectural debt could possibly influence the code design negatively. While the flawed code design may be easy to see, the actual underlying TD could be near impossible to identify. This needs to be accounted when identifying TD and therefore TD identification should be done for every part of the development process[S5]. While for product-based development, the whole product lifecycle is accompanied and measurements can be taken during the whole process, this does not have to be the case for project-based development. Because of this, even while the action taken to identify TD are not necessarily different, the practical result can differ.

There is evidence for differences in **TD measurement**. In an empirical study of Yli-Huumoo et al., teams which follow a rather project-based approach tend to implement TD measurement, while product-based teams often neglect this step[S12]. It is important to note that this anomaly seems to be correlating with the team structure. Teams which take actions to implement TD measurement hand over the responsibility for this to their team manager, a role which was only present for project-based teams in this study. A difference between product-based and project-based teams could therefore be coincidental.

For **TD monitoring** and **TD prioritization** there is no apparent connection between the development environment and the implementation of these steps. In general, teams have a tendency to neglect separate guidelines or calculations for TD prioritization and instead prioritize them in the same process as other action items[S1]. This leads to a higher impact of time- or resource-related pressure lasting on the development team. When a customer requirement or an important feature is upcoming, TD items often have a lower priority[S1].

In theory, **TD communication** should receive greater focus in project-based teams. When working with a non-technical customer, a clear TD communication could be integral for later repayment. However, a clear statement regarding this assumption is not possible with the current studies as this aspect was not widely explored.
When examining **TD prevention** there is a low influence of project circumstances and methods in general[S9]. This is also the case for the development environment.

As stated in the previous section discussing TD concepts, **TD documentation** has to be thoroughly conducted for project-based development in case of exchange of the team[S2][S6].
And finally, **TD repayment** actions generally differ depending on the circumstances like system size and age[S8]. But there are no findings to support the assumption that particularly the development environment is an influential factor to this.

In summary, while most TD concepts, regardless of the development environment, are quite similar, there are certain attributes such as TD causes and visibility which differ. This has direct implications for TDM. And when examining TDM steps as isolated as possible, there are still differences noticeable depending on multiple factors, including the development environment. TDM steps which differ the most are TD measurement and repayment.

5. DISCUSSION

5.1 Answer to research questions

RQ1 How do TD concepts differ between product- and project-based development?
As the results show, the amount of differences between the two development environments depend on the observed TD concept. Most *TD causes* occur in both software development environments and mostly differ in detail, with inappropriate planning and fixed budgets being the only exceptions found during our study.
For *TD consequences* there are no apparent differences. *TD visibility* differs the most between the development environments as visibility of TD is decreased in project-based environment due the fluctuation of project members or even full teams leaving a project.

RQ2 How does TDM change in product- vs project-based development?
There are indeed TDM steps which are generally different, e.g. TD measurement and TD repayment. However, there is a gap left. Whether the differences in TD measurement are connected to team structure or the development environment cannot be answered undoubtedly. Similarly, TD repayment is different depending on multiple process circumstances and therefore a correlation to the development environment could not be isolated[S8].

Additionally, for TD communication and documentation we had assumptions that project-based teams have a higher reward when implementing these steps, but these assumptions could not be proven nor disproven in the scope of this paper.

5.2 Implication for practitioners/researchers

When reviewing the results of our systematic literature review, it becomes apparent that multiple assumptions could not be clearly proven or disproven. This is mainly due to lack of data to support the necessary examination. Therefore, one implication for future research is that an exploratory study focusing on the influence of development environments should be conducted. A possible approach could be an industrial case study containing multiple cases with teams, working with either development environment, recording the TDM process for each team individually. This could lead to a clearer picture of the influence of development environments on the frequency of certain TD causes, the general TD visibility and the design of TDM.

Meanwhile, an implication for practitioners should be a general understanding of the influence of different process circumstances, including the development environment, on TD and TDM. More precisely, this includes a higher awareness of certain TD causes which are naturally inclined to occur within one development environment, such as *inappropriate planning* for product-based development or shortcuts motivated by fixed budgets for project-based development. Further, there are TDM steps which possibly differ in implementation. Namely, these are TD measurement and repayment. TDM steps such as TD communication and documentation could benefit from a greater focus in implementation, when working project-based. It is highly recommended to reflect these differences when approaching TDM for either development environment.

5.3 Internal validity

Many of the reviewed literature did not explicitly specify the development environment of the survey participants or observed case studies. Therefore, we established criteria to map studies to either project-based development, product-based development or mixed approaches. In some cases, the information provided lead to a well-founded assumption, but the amount of information was always limited and more profound insights into the circumstances could result in a different mapping.

Naturally, when conduction a systematic literature review, we are depending on the validity of the literature. By excluding literature which was not peer reviewed, we focused on papers with a higher probability to be valid. Although, it should be noted that for some surveys and case studies, the number of participants and subjects to the case studies were not especially large. For example, the majority of case studies include less than 10 teams, which in total results in less than 1000 participants. This limits the meaningfulness of the collected information.

5.4 External validity

As indicated in section 5.3, the overall scope of development environments studied may not be large enough to allow applying the results to the entire software development industry. As many of the reviewed papers are case studies observing a concrete development environment with all of its circumstantial dependencies, generalising the results and transferring the learnings to other projects may not be beneficial in every case, because the circumstances could differ too substantially. By including papers from 2011 to 2021 that observe development environments from years ago, todays evolution of TDM is not fully reflected. Although our results should be applicable to current projects, there may be new approaches to TDM which could be more applicable or beneficial.

6. CONCLUSION

While there is a great focus on the concepts of TD and TDM in the current scientific discussion, it is difficult to make clear statements about the effectiveness and influence of different circumstances surrounding the development process on TD frequency and the efficiency of TDM. The goal of this paper was to isolate the influence of the development environment, particularly the difference between product-based and project-based development environments, on TD concepts and TDM. Therefore, we conducted a systematic literature review, examining 12 examples of scientific literature, mostly case studies and surveys, focusing on the differences in the development environment. To compare different case studies appropriately, we first had to map those studies to project- and product-based approaches by answering a catalog of criteria for each paper. Although, limited by the amount of papers, which discuss this topic in particular, and the vagueness which comes naturally when mapping studies on such a small amount of information, our results suggest that there are no TD types only specific to one project-mode. When focusing on TDM, data shows that TD measurement and TD repayment are the two actions which differ most depending on the development environment. However, the influence on TD measurement could not be validated to be independent of other circumstances, such as team composition. The assumptions that TD communication/documentation would impact teams more if there were part of a project-based environment could not be validated or refuted. Because of our results there is a clear implication for researchers to conduct further research, ideally this would be a study focusing on industrial cases by monitoring TDM of teams with different development environments. Furthermore, more research on the implementation of single TDM actions in an industrial context would provide helpful insights into the effectiveness of different approaches to a particular step. This could lead to more precise proposals and guidelines on how to effectively implement TDM.

7. REFERENCES

[1] Areti Ampatzoglou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou. The financial aspect of managing technical debt: A systematic literature review. *Information and Software Technology*, 64:52–73, 2015.

[2] Barbara Kitchenham. Guidelines for performing systematic literature reviews in software engineering. 2007.

[3] W. Cunningham. The wycash portfolio management system. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '92, pages 29–30, New York, NY, USA, 1992. Association for Computing Machinery.

[4] J. Humble. Unit 1. introduction to the product lifecycle. `https://speakerdeck.com/jezhumbleucb/unit-1-introduction-to-the-product-lifecycle`, 2020. Accessed: 2022-05-27.

[5] M. Philip and Y. Thirion. From project to product. In P. Gregory and P. Kruchten, editors, *Agile Processes in Software Engineering and Extreme Programming – Workshops*, pages 207–212, Cham, 2021. Springer International Publishing.

[6] C. Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, New York, NY, USA, 2014. Association for Computing Machinery.

[7] Zengyang Li, Paris Avgeriou, and Peng Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

APPENDIX

A. DATABASE SEARCH QUERIES

A.1 ACM

AllField:(("technical debt management")) AND (Title:(project) OR "project-based" OR "project management" OR Title:(product) OR "product-based" OR "product management")

A.2 Scopus

TITLE-ABS-KEY ("technical debt management") AND (TITLE (project) OR "project-based" OR "project management" OR TITLE (product) OR "product-based" OR "product management")

A.3 IEEE

("All Metadata": "technical debt management" AND ("Document Title": "project" OR Search$_All$: "project−based" ORSearch$_All$: "projectmanagement" ORtitle : "product" ORSearch$_All$: "product − based" ORSearch$_All$: "productmanagement"))

B. Selected Studies

[1] Z. Codabux and B. Williams. Managing technical debt: An industrial case study. In *Proceedings of the 4th International Workshop on Managing Technical Debt*, MTD '13, pages 8–15. IEEE Press, 2013.

[2] H. Ghanbari. Seeking technical debt in critical software development projects: An exploratory field study. In *2016 49th Hawaii International Conference on System Sciences (HICSS)*, pages 5407–5416, 2016.

[3] Y. Guo, C. Seaman, and F. Da Silva. Costs and obstacles encountered in technical debt management – a case study. *Journal of Systems and Software*, 120:156–169, 2016.

[4] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Da Silva, A. Santos, and C. Siebra. Tracking technical debt - an exploratory case study. *IEEE International Conference on Software Maintenance, ICSM*, 2011.

[5] S. Malakuti and J. Heuschkel. The need for holistic technical debt management across the value stream: Lessons learnt and open challenges. In *2021 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 109–113, 2021.

[6] M. Njima and S. Demeyer. Value-based technical debt management: An exploratory case study in start-ups and scale-ups. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on Software-Intensive Business: Start-Ups, Platforms, and Ecosystems*, IWSiB 2019, pages 54–59, New York, NY, USA, 2019. Association for Computing Machinery.

[7] F. Oliveira, A. Goldman, and V. Santos. Managing technical debt in software projects using scrum: An action research. *Proceedings - 2015 Agile Conference, Agile 2015*, 2015.

[8] B. Pérez, C. Castellanos, D. Correal, N. Rios, S. Freire, R. Sp\'inola, and C. Seaman. What are the practices used by software practitioners on technical debt payment: Results from an international family of surveys. In *Proceedings of the 3rd International Conference on Technical Debt*, TechDebt '20, pages 103–112, New York, NY, USA, 2020. Association for Computing Machinery.

[9] N. Rios, S. Freire, B. Perez, C. Castellanos, D. Correal, M. Mendonca, D. Falessi, C. Izurieta, C. B. Seaman, and R. O. Spinola. On the relationship between technical debt management and process models. *IEEE Software, title=On the Relationship Between Technical Debt Management and Process Models*, 38(5):56–64, 2021.

[10] C. A. Siebra, G. S. Tonin, F. Q. B. Da Silva, R. G. Oliveira, L. C. Antonio, R. C. G. Miranda, and A. L. M. Santos. Managing technical debt in practice: An industrial report. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 247–250, 2012.

[11] A. Sousa, L. Rocha, R. Britto, Z. Gong, and F. Lyu. Technical debt in large-scale distributed projects: An industrial case study. *Proceedings - 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021*, 2021.

[12] J. Yli-Huumo, A. Maglyas, and K. Smolander. How do software development teams manage technical debt? – an empirical study. *Journal of Systems and Software*, 120:195–218, 2016.

C. MAPPING RESULTS PER PAPER

Managing Technical Debt: An Industrial Case Study[S1]

(1) While the paper does not explicitly state the observed development environment, it is mentioned that the survey had about 250 participants from 28 different teams. Therefore, it is assumed that this paper contains mixed development environments.

⇒Mixed

Seeking Technical Debt in Critical Software Development Projects: An Exploratory Field Study[S2]

(1) It is explicitly stated that the multiple teams observed are working in highly different development environments.

⇒Mixed

Costs and obstacles encountered in technical debt management[S3]

(1) No.

(2) Yes, a project lifetime of 17 sprints was defined.

(3) The presence of a single customer is stated.

(4) Whether the whole lifecycle is accompanied was not clearly stated.

⇒Project-based development, because of criteria (2) and (3).

Tracking technical debt - An exploratory case study[S4]

(1) No.

(2) No, the project ran for the whole product evolution process.

(3) No single customer was stated.

(4) The product was developed by the same team pre- and post-release.

⇒Product-based development, because of criteria (2), (3) and (4).

The Need for Holistic Technical Debt Management across the Value Stream: Lessons Learnt and Open Challenge[S5]

(1) No.

(2) Not explicitly stated.

(3) Not explicitly stated.

(4) Not explicitly stated.

⇒Unknown, because not enough information on the project is provided to map this case study.

Value-Based Technical Debt Management: An Exploratory Case Study in Start-Ups and Scale-Ups[S6]

(1) Yes, it is made clear that the participants where start-ups developing a single product or service.

⇒Product-based development.

Managing Technical Debt in Software Projects Using Scrum: An Action Research[S7]

(1) No.

(2) No.

(3) Project 2 (out of 2) had a single customer.

(4) Project 1 was a specific adjustment of a finished product for one use case.

⇒Project-based development, because of criterion (3) in case of project 2 and criterion (4) in case of project 1.

What Are the Practices Used by Software Practitioners on Technical Debt Payment: Results from an International Family of Surveys[S8]

(1) While the paper does not explicitly state the observed development environment, it is stated that this is an open survey with 432 participants from multiple companies with different demographics.

⇒Mixed

On the Relationship Between Technical Debt Management and Process Models[S9]

(1) While the paper does not explicitly state the observed development environment, the survey contains a question about the development approach of the corresponding project and groups its results by development modes. Therefore, multiple different development environments are reflected by the participants of this survey.

⇒Mixed

Managing technical debt in practice: An industrial report[S10]

(1) No.

(2) No.

(3) Multiple possible customers exist.

(4) One team is accompanying the whole lifecycle.

⇒Product-based development, because of criteria (2), (3) and (4).

Technical Debt in Large-Scale Distributed Projects: An Industrial Case Study[S11]

(1) Multiple teams work on one product but in independently carried out projects. A project-based approach is therefore clearly stated.

⇒Project-based development

How do software development teams manage technical debt?[S12]

(1) Table 1 in this journal article contains a list of all teams depicted in this case. Those teams are working product-based as well as project-based.

⇒Mixed

Property-Based Testing : Application fields, challenges and new approaches

Sara Prifti
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
sara.prifti@rwth-aachen.de

Sandro Schulte
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
sandro.schulte@rwth-aachen.de

ABSTRACT

Property-based testing (PBT) is a well-known testing method in theory and has found usage since the famous QuickCheck library release in 1999. It is different from standard example-based testing techniques, as it randomizes and generalizes test cases based on property specifications. This allows coverage of unconsidered edge cases and equivalence classes. In addition, PBT can be combined with other well-known techniques like combinatorial testing to further improve effectiveness. Yet it is not as widely used in the industry as other testing approaches, despite tooling support for popular programming languages.

In this paper, we analyze the characteristics of real-world projects applying PBT to identify challenges in its application. Therefore we queried GitHub to find relevant projects that depend on specific PBT frameworks. Based on the findings, we elaborate on the discovered challenges and present approaches from the literature to solve them.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*productivity, programming teams, software configuration management*

Keywords

property-based testing, software quality assurance, QuickCheck, GitHub, real-world testing

1. INTRODUCTION

Software testing is a method to provide software quality and is usually realized by writing test cases for a set of inputs to check if the output differs from the space of expected results. Example-based testing is one of the many techniques in which the software behavior is tested towards concrete scenarios. However, the example-based tests do not cover the complete range of results since the number of possible generated cases might be out of scope for the developer. Thus, it would be impossible to find faults when some inputs that might be important are not considered. Therefore, another technique called boundary-value analysis can be used. Instead of testing simply inside a certain range of inputs, this method examines the system's behavior towards the input boundaries. Yet, the chosen values for testing are predefined and pre-selected, which means that the problem of not covering important test cases still arises [9, 16, 11].

For these reasons, this paper analyzes property-based testing (PBT), a testing technique that deals with these drawbacks. The PBT tools generate random inputs automatically to test the properties of the system under test that are formalized by the user beforehand. Quickcheck is the first tool for property-based testing, which led to the development of other approaches and improved extensions for PBT. It is used to test whether Haskell-written properties hold up against these randomly generated inputs. Property testing helps developers to reduce the size of test code and at the same time increases the likelihood of encountering a bug by testing cases that developers are unaware of [25]. For example, when wanting to test whether a system authenticates the user correctly, PBT can be used to test the authentication property that is specified based on the source code implementation [10].

However, because of problems that arise during property-based testing related to the range of generated inputs, expected results and difficulty in property definition, we considered three research questions.

RQ1: Is the usage of PBT limited to specific application fields?

RQ2: How popular is PBT in open-source repositories?

RQ3: What are the challenges that might prohibit the usage of PBT?

This paper is organized as in the following. Section two describes what property-based testing is, how results are produced and verified, and how PBT is combined with other testing approaches. The third section explains where PBT is used and where it is applied. Also, in this section, a short code is implemented that helps with finding the most rated projects related to the PBT on GitHub [18]. Afterwards, section four illustrates an example of possible PBT problems that arise when deriving properties from business rule models, followed by section five presenting three new approaches, that try to deal with those problems.

SWC Seminar 2022 RWTH Aachen University, Germany.

2. FOUNDATIONS OF PROPERTY-BASED TESTING

In testing, we always have to cope with the *combinatorial explosion* of input complexity [4]. It is impossible to test every input of a given unit under test (UUT). *Property-based testing* (PBT) approaches this by defining properties describing the allowed range of inputs and the expected outcomes [6]. Since concrete in- and output values are required for testing, PBT frameworks have to generate test cases from property descriptions. This can be achieved by providing the properties in a machine-readable format.

With PBT, we can cover all valid input combinations with a single property description [6]. Test cases are automatically and randomly generated from the allowed range of inputs. Manual creation of relevant test cases is no longer required. In addition, missed edge cases or equivalence classes can be detected. The random test cases have a chance to cover them. However, we have to provide a correct and complete set of properties to ensure adequate testing. Every specification and every possible input must be considered when they are defined. Further challenges are discussed in section 4.

PBT is a *generative approach* and has similarities to Fuzz Testing [12] and example tables in Behavior-Driven Development (BDD) [24]. It can be used for unit, component, and system testing. Further functional and non-functional testing is possible. Since we formulate properties based on the UUT specification, PBT realizes a *black-box testing* approach. The internal structure of the UUT is not considered. Finally, PBT is not restricted to specific project types.

2.1 Defining Properties

The first step for applying property-based testing is the definition of properties [6]. They can be formulated as follows:

```
for all (x, y, ...)
satisfying precondition (x, y, ...)
property (x, y, ...) holds
```

The definition consists of two parts. Firstly all available input variables are defined. Secondly, a valid input range for which the property must hold is specified with a precondition. Lastly, the property that holds for this input configuration is described. The precondition and the property are denoted as a boolean expression. Through this, the input and output of a UUT are connected.

We use an addition operation as an example to show the concrete difference between example-based testing and PBT. The function `add(x,y)` adds two integer numbers. In standard example-based tests this could result in the test case:

```
Given (3, 2)
When I add them
Then I expect 5
```

With PBT, we can use the mathematical properties of the addition operations instead:

- Commutativity

```
for all (int x, int y)
satisfying precondition (true)
property (add(x, y) equals
     add(y, x)) holds
```

- Associativity

```
for all (int x)
satisfying precondition (x greater 0)
property (add(add(x, 1), 1) equals
     add(x, 2)) holds
```

- ...

These properties are more general than the concrete example before. We also notice that properties can contain variable amounts of inputs, constant values, or too specific preconditions. This can result in less test coverage and therefore a lower test quality. For instance, we can describe the associativity property as

```
for all (int x, int y, int z)
satisfying precondition (true)
property (add(add(x, y), z) equals
     add(x, add(y, z))) holds
```

to cover more applicable cases.

2.2 Test Procedure

After the property definition, we can insert them into the testing framework. The already mentioned QuickCheck library for Haskell has set the foundations of the testing procedure [6]. We use it as an example to explain the general structure of every PBT framework. Figure 1 shows the four parts of a QuickCheck. A Checker API as a control structure, a generator for providing concrete test inputs, the test runner and a shrinker to provide human-understandable results.

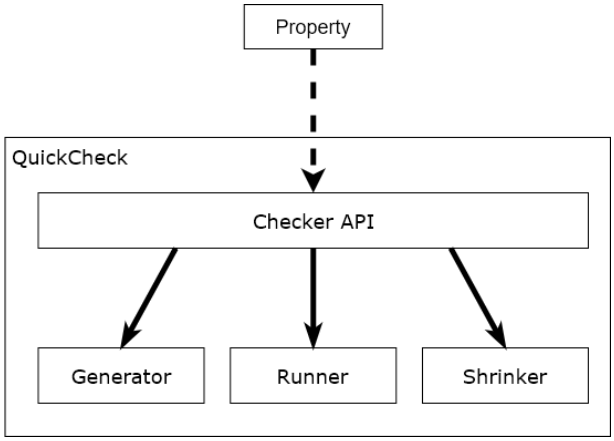

Figure 1: Structure of the QuickCheck property-based testing framework.

Each property is handed to the *Checker API*. It generates random input sequences with the help of the *generator* and based on the input specification of each property. The given logical formula needs to be evaluated to check if an input sequence is valid. Then each valid input sequence will assemble a test case. So the number of test cases is controlled by the number of generated input sequences. The generator can use a seeding mechanism to provide reproducibility [8]. A seed is an initialization value for the randomization function. Previous test runs can therefore be repeated by providing the same seed.

All input sequences are then passed to the *test runner*. Based on the property specification, it executes the UUT and other helper functions with the given sequence. In the end, the logical formula of the property can be evaluated. If it holds, the test case passes and the next one is executed. If not, an error is discovered and the test case fails.

Since the generated input can be very complex, the result of a failed test case is often challenging to comprehend. Particular difficult situations occur when lists are required as input. The generator will produce arbitrary large lists filled with many different elements. If a test case reports an error, the user cannot quickly understand which part of the list causes the problem. Therefore, a shrinking of the generated input sequence should be applied. This step is optional and not used in every PBT framework since it only improves usability. So the testing framework can be simplified if complex input sequences cannot occur.

The *shrinker* module is executed if a test case fails. It takes the failing input sequence and applies a simplification step. This simplification requires different algorithms and heuristics, like reducing the list size by one. If the new input sequence fails again, we have found a valid reduction and can repeat the process. If not, we have already found the smallest input sequence and can output this to the user. It is possible to use multiple and different simplification procedures [14]. A failing simplification step can then be replaced by another one. To find the smallest failing input sequence, all possible combinations of simplifications have to be tested. This results in a tree of input sequences shown in figure 2.

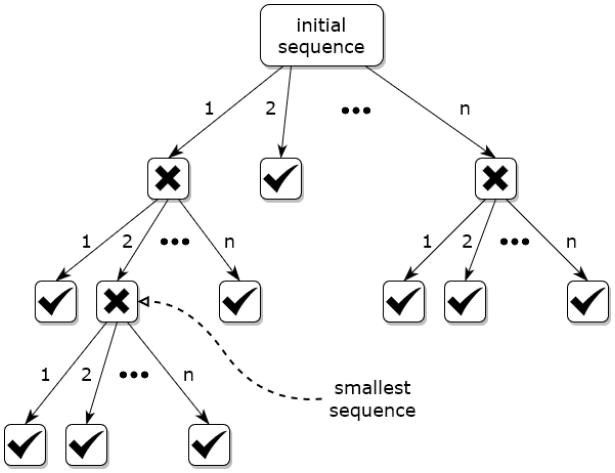

Figure 2: Tree of input sequences when shrinking is applied. Nodes represent failing (cross) and succeeding (checkmark) input sequences of a test case. Each edge applies a specific simplification procedure according to the edge label.

The effectiveness of the shrinking heavily depends on the used simplification procedures [14]. We could miss smaller configurations if the procedures reduce the input too much in one step or combine different simplifications. However, if they change the input only slightly and are numerous, the tree of possible input reductions can get huge. In addition, complex data structures can be difficult to shrink. Therefore, usually only basic data types are considered in simplification steps. Alternatively, user-provided procedures can be applied to improve shrinking.

At the end of the testing procedure, the *Checker API* combines all results and returns them to the user. A property can be marked as passed if it has no failing test cases. For all other properties, the failing and shrunk input sequences are returned.

2.3 Combination with other Testing Techniques

Different testing techniques are often applied simultaneously to improve software quality even more [3]. Beside using other techniques separately from PBT, we can combine them with PBT. This will increase the complexity of the testing framework but also make all their benefits available at the same time. In this section, we have a closer look at boundary value, equivalence class and combinatorial testing as the most popular alternative approaches.

Boundary value testing [3] is a common technique to test special input values. We can modify our generator module to always produce input sequences with given boundary values in addition to the random test cases. As a result, we can guarantee that the properties are tested with all recognized special cases.

As an extension to edge case consideration, *equivalence class testing* [3] splits the whole input domain into different parts. Concrete input sequences from the resulting classes must have similar properties. So the consideration of equivalence classes can help to define properties. We can also use equivalence classes to modify the generator further. By ensuring that input sequences are generated in every given equivalence class, we can ensure that the domain is tested more thoroughly. It is no longer possible to miss previously known equivalence classes in a test run.

Finally, *combinatorial testing* [3] ensures the coverage of different input value combinations. This is important if multiple parameters interact in the execution of the UUT body. Combining combinatorial testing with PBT will also result in a modification of the generator module. We require that specific combinations of input values are generated. Since boundary values and equivalence classes can be incorporated into the generation, as seen before, this modification is possible.

We can observe that all combinations require a modification of the generator module and result in a more specific set of test cases. Consequently, fewer test cases need to be generated to cover all relevant input sequences. Combined with optimized shrinking and strong properties, this ensures efficient and effective testing.

3. APPLICATION OF PBT IN REAL-WORLD PROJECTS

The application of PBT in the real world requires a good testing infrastructure. Without frameworks and tool support, the testing of software projects is infeasible. Usually, these tools are developed externally and can be used in different projects. There exist many PBT frameworks for popular programming languages like:

Java JUnit-Quickcheck

C++ RapidCheck

.NET FsCheck*

JavaScript fast-check*

Ruby rubycheck

Python Hypothesis

Scala ScalaCheck

PHP PhpQuickCheck

Haskell QuickCheck

This list is incomplete and shows only a subset of relevant projects. Nevertheless, it represents a wide range of possible usage scenarios like Website development in PHP, academic usage in Haskell, or end-user software in C++. The two testing frameworks marked with an asterisk are later used for a deeper analysis.

Besides using predefined infrastructure, large software projects sometimes implement their own frameworks or adapt existing ones to their needs. Two popular examples in the context of PBT are Apache Flink [8] and the Robot Operating System (ROS) [23]. The first project is a stream processing framework and introduced a completely new testing framework for Java called FlinkCheck. This testing framework is explicitly fitted to Apache Flink but can also be applied in other Java projects. It uses a temporal logic to consider time in property definition and test case generation. ROS, on the other hand, is a framework for the development of robotic software. Since it uses distributed and reusable components written in different programming languages, traditional frameworks can only provide unit and component testing capabilities. However, the problematic task of component integration remains untested. Hence PBT was adapted to provide testing capabilities for this task.

The ROS and Apache Flink examples show that PBT can be applied in very different contexts. In the following, we analyze application fields of PBT apart from stream processing and robot development. Therefore, projects dependent on specific PBT frameworks are searched, categorized, and listed.

3.1 Methodology

To find further application fields of PBT, we queried *GitHub* for dependencies. GitHub natively provides an insight on all dependents of a repository. Thus we can find all GitHub projects that rely on PBT frameworks published on GitHub. Since the list of dependents cannot be sorted or filtered on the GitHub webpage, we use the GHTOPDEP project [2] for that purpose.

GHTOPDEP is a python command-line tool. It requires a public_repo token to access all public repositories on GitHub. Then it queries GitHub for all dependents of a given repository URL using the insight feature. The results are given in a table and sorted by stars. Each table entry contains the repository URL and the number of stars. Further, the resulting table can be clipped after a specified amount of entries and projects with too few stars can be excluded. Beside the sorted table, additional information about all dependent public repositories is displayed. This includes the total quantity of found repositories and the quantity of those with at least one star.

We are using stars as a sorting criterion since they are the best available metric for project popularity and relevance. Every registered user account can give stars freely to a GitHub repository. They can be used as a bookmark or to show appreciation for a project. Consequently they do not directly reflect usage and can be manipulated with fake accounts. Still, they display an interaction with a project, are difficult to manipulate in large amounts, and also reflect the popularity of every project type. Alternatives like the dependent count fail to measure projects that are not very reusable (e.g. end-user software), while derived metrics would require complex weighting and normalization.

In the following section, we evaluate two PBT frameworks for different programming languages with the introduced tool. It is set to display a maximum of 50 repositories with at least 1 star. Because we are only interested in applications fields of PBT, neither the restriction to public GitHub projects rated by stars nor the small sample size of two frameworks has a strong impact on the results. If we find a highly rated project for a specific application field, we can assume that PBT was at least considered in this context. So PBT should also be a valid consideration for other projects in this application field because they often have similar challenges, design decisions, or programming languages. Only the extent to which PBT is used cannot be inferred from a dependency.

3.2 Results

The first analyzed framework is fast-check [22] for JavaScript. It was first announced in 2017 and then finally released in October 2018 with version 1.0.0. It is still in active development and currently available in version 3.0.0. The framework is primarily written in TypeScript and publicly available on GitHub. With a count of 3100 stars and over 600 thousand downloads per month, the project already has a considerable amount of popularity.

fast-check implements the full PBT feature set including shrinking. With the help of seeding, it also allows to reproduce previously executed tests. Beside the basic PBT implementation, no other testing approaches or fundamental modifications to the property-based approach are provided with this framework.

Table 1 shows the command-line tool output for fast-check. The table only shows the first ten entries of the result and is extended by a short description of the detected projects. We can directly observe repositories with high star counts above 20 thousand like Jest or Ramda. In comparison to fast-check, these are highly popular software projects. Even the last table entry has a considerable amount of relevance with 6500 stars.

When considering the complete tool output, more than 30 projects with at least 1000 stars are listed. The lowest-rated project in position 50 still has more than 500 stars. In total, 9430 public repositories were found with 2510 of them having at least one star.

If we focus on the descriptions, we detect a variety of different project types. There are testing frameworks, cloud and distributed systems development software, digital currency libraries, and even software for developing user interfaces. Some application fields occur multiple times, like the functional programming libraries. However, no clear focus on one type of project can be detected in general. In addition, all results can be considered typical for JavaScript since this language is mostly used to develop dynamic and interactive web content, mobile apps, and server applications.

FsCheck [20] is the second analyzed PBT tool and enables testing of .NET software. It was first announced in 2014 and released in version 1.0.0 later that year. It is still in active development and currently available in version 2.16.5. The software is mostly written in F# and its source code is publicly available on GitHub. With a count of 977 stars, it is less popular than fast-check, although released earlier.

Table 1: Top 10 dependents on fast-check sorted by stars.

repository	stars	description
facebook/jest	39K	JavaScript Testing Framework
ramda/ramda	22K	Functional Programming for JavaScript
OpenZeppelin/openzeppelin-contracts	18K	Smart Contract Development Kit
jasmine/jasmine	15K	JavaScript Testing Framework
trufflesuite/truffle	13K	Ethereum Development Kit
tinymce/tinymce	11K	Text Editor
aws/aws-cdk	8.7K	Cloud Computing Development Kit
MetaMask/metamask-extension	8.0K	Crypto Wallet Browser Extension
gcanti/fp-ts	7.8K	Functional Programming for TypeScript
adobe/react-spectrum	6.5K	User Interface Development for Adobe Software

Table 2: Top 10 dependents on FsCheck sorted by stars.

repository	stars	description
dotnet/runtime	9.2K	.NET Software Development Framework
akkadotnet/akka.net	4.1K	Distributed Systems Development Kit
commandlineparser/commandline	3.5K	Commandline Parser
dotnet/fsharp	3.1K	F# Programming Language
microsoft/onefuzz	2.6K	Fuzz Testing Framework for .NET
fsharp/fsharp	2.2K	F# Programming Language (old)
MetacoSA/NBitcoin	1.7K	Bitcoin library
PragmaticFlow/NBomber	1.5K	Load Testing Framework for .NET
NeutroniumCore/Neutronium	1.3K	Library for .NET programming via HTML, CSS and javascript
bryanedds/Nu	823	Functional Game Engine

Since FsCheck is a port of Haskell's QuickCheck library, it implements the complete PBT feature set. Similar to fast-check, a seeding mechanism for reproducibility is provided, but no additional modifications or extensions are introduced.

The command-line tool could detect a total of 1012 repositories dependent on FsCheck, with 327 of them having at least one star. The complete table ranges from projects with 40 stars to projects with 9200 stars. More than 30 projects have at least 100 stars. So beside the smaller star count of FsCheck itself, the number and popularity of dependent projects are also much smaller than for the fast-check example.

Since .NET software platform is mostly used for the development of application software for end-users, it has a different focus than JavaScript and is still relevant for the analysis. A closer view on the top 10 highest rated repositories for FsCheck in table 2 reveals some additional application fields. Parsing, Game Engine development, and programming languages are new examples where PBT is applied. Nevertheless, some previously detected use cases, like testing frameworks and digital currency tools, occur again.

In summary, we can detect usage or consideration of property-based testing in thousands of projects. Moreover, popular projects with up to 40000 GitHub stars are also dependent on PBT frameworks. Finally, the property-based approach does not seem to be limited to specific application fields. However, compared to traditional example-based testing, both analyzed frameworks are still very unimportant. Alternatives like the BDD framework Mocha [19] in JavaScript or the non-PBT framework xUnit [21] in .NET have a much higher star count and are used in vastly more projects. Figure 3 shows the differences in popularity between all mentioned projects. Especially the dependent count is a significant indicator of popularity since testing frameworks are designed for reuse. So property-based testing must have other challenges that prevent developers from using this approach.

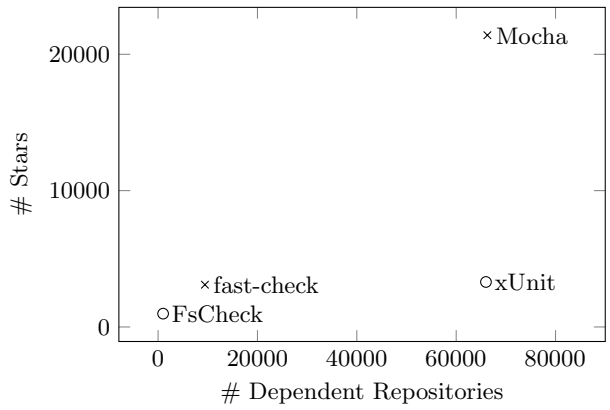

Figure 3: Number of stars and dependent public GitHub repositories on different PBT and non-PBT testing frameworks.

4. CHALLENGES

To discuss challenges in the application of PBT, this section describes a specific example of how property-based testing is used for testing business rules [5].

The relational database management system (RDBMS) has been used before by many applications to store data since it offers the possibility to define data constraints that keep data structured and consistent. Business rules represent the consistency checks of these data constraints and ensure that the application is well-performing. For their validation, there are automated tests for within and outside

the input bounds generated using QuickCheck. To understand how this works an ordering system example, taken from Laura M.Castro [5] will be explained. As given in [5] figure 4 illustrates an Entity-Relationship Diagram (ERD) between the three system entities, customer, product, and order.

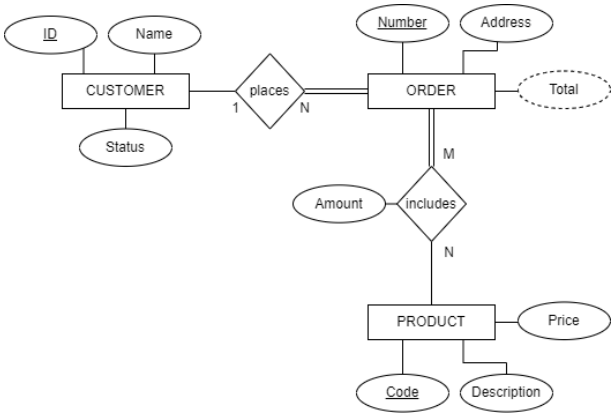

Figure 4: ERD between customer, product and order entities [5]

When translating the diagram in the figure 4 into RDBMS there are constraints implemented, such as the entities should contain primary keys, the relationships between the entities are represented as relations with one or more keys, the amount or the product price should not be empty, and so on. Other constraints that rely on the system operation, like how a customer gets the gold status or what featured products can be bought only by gold customers, are the so called business rules. To test if the system has implemented these rules correctly they are translated into SQL queries. Below is an example of a business rule written in the SQL query, which states that the customer can purchase a featured product only if the customer has the gold status [5].

```
--Business rule:

SELECT customer.id
  FROM customer, order, order_products
 WHERE customer.id = order.customer_id
   AND order.number = order_products.
    order_number
   AND customer.status <> 'gold'
   AND order_products.products_code
    IN <featured product list>
```

Listing 1: Business rule in SQL query

Assume that the customer obtains the gold status after placing at least five orders. Since the system allows modifying or canceling the order, it presents the possibility of violating the business rule. When achieving the gold status, the customer might place an order which contains a featured product and afterwards cancel the other four placed orders. In this situation the customer would have a special product and a non-gold status. Since QuickCheck randomizes tests in the above-described case, it could happen that it is continuously tested only for placing an order with random non-existing products or the orders are being canceled [5]. In this way there will not be any new testing results produced.

To overcome this challenge and produce more test cases, state-machine testing can be applied. This technique uses the knowledge of past test results to generate new ones. All data stored in the database at a particular moment is what makes a state, while the path from one state to another is the transition of the state machine. Using only one state for the whole database and updating only that state continuously does not provide the possibility of finding errors. Therefore, when testing it is important to keep only enough data in the current state to generate all the possible new states and transitions. The input data is already contained in the database or it is randomized, meaning that new information is added to the database and can be re-used for the other possible test cases [5]. In this example the author uses Quviq's Quickcheck which generates programs written in the Erlang, a programming language for real-time systems, that provide high availability [5, 7].

Firstly, an abstract state with an empty customer and product information is defined. The customer field contains the customer id and the orders placed by that customer, while the product field will contain the list of purchased products [5].

```
initial_state() ->
  #shop{customers = [] ,
        product = []} .

#customer {id, orders=[] }
```

Listing 2: Definition of the first state

The *next_state* function shows how the interaction between the state and the chosen data for testing affects the database. Firstly the newly created customer is added to the list of customers with an id and places orders on that id. Then the *next_state* function updates the state based on the information about the customer and the product. If the information was present in the abstract state, it means that the data was newly produced and the database is updated by adding the new orders placed in the customer's information, otherwise the state does not change [5].

```
next_state(State, CallResult,
 {call, ?MODULE, new_customer, [Name]}
 )->
  NewCustomer = #customer {id = CallResult
    },
  State#shop{
     customer = [NewCustomer] ++
                   State#shop.customers};

next_state(State, CallResult,
 {call, ?MODULE, place_customer, [Id,
    ProductCode]}
 )->
 case existing(ProductCode, State#shop.
    products) of
  true ->
   case get_customer(State#shop.customers,
    Id) of
    not_found -> State;
     Customer ->
      NewOrders = [CallResults] ++
                   Customer#customer.
    orders,
       State#shop{
        customer =
          [Customer#customerorder =
    NewOrders] ++
            delete_customers(S#shop.
    customers,Id)]}
```

```
    end;
  false ->
    State
end;
```

Listing 3: Definition of the newly created and updated states

When updating the customer records with the new data, the previous old records are removed from the state. In this part the problem relies on the fact that after updating the state, the database also changes and there is no way of checking the consistency of the updated information with the business rules. Afterall, the original database is not saved in an internal state of the machine model. Therefore, it was proposed to check for the violation of business logic after every sequence of tests. This enables the tests to run faster and in parallel [5].

As mentioned at the beginning of this section each business rule is translated into SQL queries and is embedded in a database transaction.

```
business_rule(Connection)->
 [] == db_interface:process_query(
    Connnection,
       "SELECT id"
       "  FROM order_products NATURAL JOIN
    customer"
       " WHERE status <> 'gold' "
       "   AND code IN ( "
       ++ string:join (?GOLDEN_PRODUCTS])
                          ++" ) " )"
```

Listing 4: Embedding a business rule as part of a transaction

The idea of the proposed solution is to evaluate each requested data for testing by checking if the database invariant holds and then observe the compliance of the updated database with the business logic. Afterwards, the expected results are compared with the actual results from testing. In case of a successful test sequence, other tests are generated from QuickCheck, elseways, it shrinks the set of counter examples in case of test failures, improving testing effectiveness [5].

```
prop_business_logic() ->
 ?FORALL (Commands, commands (?MODULE),
  begin
   true = invariant(),
   {History, State, Result} =
       run_commands(?MODULE, Commands),
   PostCondition = invariant(),
   clean_up(State),
   (PostCondition == true ) and (Result ==
     ok)
  end) .
```

Listing 5: Testing the compliance of the updated database with the business logic rule

The above-described example is evaluated using a risk management information system, ARMISTICE, that contains a database schema with almost 124 data-related business rules. In such cases, when the number of business rules is large, it is suggested to have more than one state machine as a test module for every possible operation or command affecting subsets of the database schema. One reason is that the complexity of the business rules affects the size of the database. As a consequence tests require more time to be executed and also the number of bugs increases. Along with the long testing time, there is also a delay in shrinking the counter example set during the re-run of the failing tests [5].

PBT also faces challenges in web-service applications. The described approach in [1] uses FsCheck to test business rule models when deriving generators for sequences of web-service requests. This tool supports, similarly to QuickCheck, models with states and transitions, where the inputs are translated from parsed XML files to extended finite state machines. The goal in [1] is to test the incoming requests between web service applications, which in this case are the transitions of the state machines. Here a transition represents the incoming data, the saving of the web page, and the graphical user interface that is used from the FsCheck to derive the properties for testing. One problem that occurred during testing happened because of the lacking information regarding business-rule models, such as the reference attributes. For this reason, the tool reported unreal bugs or exceptions.

However, PBT has proven to be suitable when testing business rules, since predicting the possible sequence of operations that caused their failure is sometimes out of scope for a developer. They often do not hold due to the unpredictable changes in conditions, thus, generating sorts of test cases that might cause a failure helps in detecting errors [5].

5. NEW APPROACHES

In the previous chapter there were described some challenges of PBT, such as continously testing with random inputs even if no new test results were being produced, incrementation of business rules complexity followed by longer testing time, delay in the shrinking of the counter examples set or lack of information regarding business-rule models causing unreal bugs or exceptions. Therefore, in this chapter there will be presented some new approaches which deal with some of these challenges.

For example having a large space of input is crucial when testing out of the input bounds or also known as negative testing. However, this does not always assure yielding satisfactory results. An approach called targeted property-based testing (TPBT), instead of testing via random inputs, applies a search strategy for the input generation, specified by the user [15]. The user defines a search-based component that generates inputs with a higher likelihood of falsifying a property based on a utility value which measures how close the random input was to falsifying that property. It has been shown that the TPBT requires fewer test cases to find bugs than the random PBT technique.

In case of the property definition it might happen that the properties contain preconditions on their inputs, which are not always satisfied, such as the list should be sorted or must not contain duplicates. Therefore, the system can react by dropping these inputs, leaving many cases untested. For these reasons, in [13] there was proposed a new approach of PBT, namely coverage guide property-based testing (CGPT), an extension of QuickCheck called FuzzChick, which controls the information flow. FuzzChick contains a generator that produces initial inputs and mutators that are type-aware. The inputs are mutated until they do not generate new test results anymore. This approach is based on fuzz testing, a technique that inputs random bytes in a program and tests whether the system fails. As a result it covers more tests than those that generate inputs for each test [13].

Except for tools, there are also libraries created with the

purpose of using PBT. An example is the Hypothesis, a library built for Python. It has been observed to improve the scientific research results in software testing and supports test-case reduction and targeted property-based testing, where the user only specifies the goal and tests are generated towards that goal [17].

6. CONCLUSION

In this paper, property-based testing was analyzed as a software testing method as well as the characteristics of real-world projects that apply PBT to identify challenges and its application. The root of PBT starts with Haskell, a functional programming language used to define properties in the form of functions. Tools like QuickCheck use these properties to generate random inputs and compare the output with the expected result. After each positive result, the set of counterexamples becomes smaller, while testing and the generation of inputs continue until there are no new results. It is proven that PBT is useful, especially in large systems where the number of possible inputs is out of scope for the developers. Again, manually written tests do not guarantee full coverage of possible inputs and thus possible results. Additionally, not considering all the inputs increases the difficulty of finding faults. However, QuickCheck is not the only tool. With the software becoming more complex when testing properties, problems also escalated. Therefore methods such as state-machine testing, usage of SQL queries to define properties, new tools like FuzzChick, FsCheck, and libraries, such as Hypothesis, have been developed.

7. REFERENCES

[1] B. K. Aichernig and R. Schumi. Property-based testing of web services by deriving properties from business-rule models. *Software & Systems Modeling*, 18(2):889–911, 2019.

[2] R. L. Andrii Oriekhov. Ghtopdep, 2019. `https://github.com/github-tooling/ghtopdep`, last accessed on 2022-07-06.

[3] A. Bhat and S. Quadri. Equivalence class partitioning and boundary value analysis-a review. In *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 1557–1562. IEEE, 2015.

[4] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *Proc. of the Intl. Conf. on Software Testing Analysis & Review*. Citeseer, 1998.

[5] L. M. Castro. Advanced management of data integrity: property-based testing for business rules. *Journal of Intelligent Information Systems*, 44(3):355–380, 2015.

[6] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000.

[7] Erlang. `https://www.erlang.org/`, last accessed on 2022-07-10.

[8] C. V. Espinosa, E. Martin-Martin, A. Riesco, and J. Rodríguez-Hortalá. Flinkcheck: property-based testing for apache flink. *IEEE Access*, 7:150369–150382, 2019.

[9] L. Ferretti. `https://www.codemotion.com/magazine/devops/qa-testing/property-based-testing-2/`, February 12, 2020.

[10] G. Fink and M. Bishop. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74–80, 1997.

[11] T. Hamilton. `https://www.guru99.com/equivalence-partitioning-boundary-value-analysis.html`, April 16, 2022.

[12] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[13] L. Lampropoulos, M. Hicks, and B. C. Pierce. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[14] F.-Y. Lo, C.-H. Chen, and Y.-P. Chen. Shrinking counterexamples in property-based testing with genetic algorithms. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2020.

[15] A. Löscher and K. Sagonas. Automating targeted property-based testing. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 70–80. IEEE, 2018.

[16] D. MACIVER. `https://increment.com/testing/in-praise-of-property-based-testing/`, August 2019.

[17] D. R. MacIver, Z. Hatfield-Dodds, et al. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 2019.

[18] Microsoft. Github, 2008. `https://github.com/`, last accessed on 2022-07-06.

[19] OpenSource. Mocha, 2011. `https://github.com/mochajs/mocha`, last accessed on 2022-07-06.

[20] OpenSource. Fscheck, 2014. `https://github.com/fscheck/FsCheck`, last accessed on 2022-07-06.

[21] OpenSource. xunit v2, 2015. `https://github.com/xunit/xunit`, last accessed on 2022-07-06.

[22] OpenSource. fast-check, 2018. `https://github.com/dubzzz/fast-check`, last accessed on 2022-07-06.

[23] A. Santos, A. Cunha, and N. Macedo. Property-based testing for the robot operating system. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 56–62, 2018.

[24] C. Solis and X. Wang. A study of the characteristics of behaviour driven development. In *2011 37th EUROMICRO conference on software engineering and advanced applications*, pages 383–387. IEEE, 2011.

[25] K. Yatoh, K. Sakamoto, F. Ishikawa, and S. Honiden. Arbitcheck: A highly automated property-based testing tool for java. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 405–412. IEEE, 2014.

Classifications of Test Oracles found in Literature

Svetoslav Apostolov
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
svetoslav.apostolov@rwth-aachen.de

Florian Braun
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
florian.maximilian.braun@rwth-aachen.de

ABSTRACT

Test oracles are an integral part of software testing to verify that the SUT (Software Under Test) behaves correctly to its specifications. Since the test oracle problem was first investigated in 1978, new approaches have been designed over time and currently the focus has shifted towards automating existing solutions. We divide the different methods into classes and look at their temporal evolution in research. For the categorisation, we looked at literature of different approaches in the context of their time periods and present the results of our analysis. Test oracles can be categorized into pseudo oracles for testing non-testable programs, test oracles based on specifications, metamorphic testing based on metamorphic relations, test oracles created by machine learning or search based learning, regression testing and assertions. These categories are defined in the context of present literature and are subject to change after further research, be it through refinement of existing or definition of new groups.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*productivity, programming teams, software configuration management*

Keywords

Test Oracles, Classification, Temporal Evolution

1. INTRODUCTION

Software testing allows for easier discovery of issues in the internal logic and confirming the successful conflict resolution thereof by checking the received output for a set input [11]. It is a key part of development and an accurate check of program reliability, though it is also costly to efficiently implement in both time and effort [5]. Choosing the appropriate input and methodology is a complex task that evolves alongside the development process [3],[5]. Maintenance costs, which are mostly delegated to testing, almost certainly surpass initial production costs because of it, given the software is supported for a certain time [14]. Because of the importance of this process, the term *test oracle* was introduced as the first step in formally categorizing different test methods [9].

In testing the test oracle takes over the role of verifying the correctness of an output for a given input and can therefore validate the behaviour of a system. To be able to confirm a software's actions the test oracle needs to be created in such a way that it is capable of knowing the right output for each possible input. This process is therefore in most cases costly in computation time and human effort [1].

Many different oracles have been newly discovered and reclassified since, with new literature on the subject being published frequently. However, this abundance of research material is insufficiently organised. Attempts to classify oracles have been made, but are either not up to date, or not universally applicable [9], [1].

RQ1: The first research question of this paper is to analyze publications on solving the test oracle problem ranging from highly specific to generally applicable solutions and separate them into distinct classes.

RQ2: In addition the second research question is to identify research trends in the field of test oracles and analyzing the temporal evolution of the interest in the classes and give a comparison.

Therefore we will analyse and classify different methods and approaches to designing a test oracle. Chapter 3 takes a deeper look at options for testing non testable programs, as well as the classes of test oracles called metamorphic testing (3.2), regression testing (3.3), specification languages (3.4), assertions (3.5), implicit test oracles (3.7) and machine learning approaches to automate the before mentioned test oracles (3.6).

After the classification we analyse a temporal evolution of research on the aforementioned classes of test oracles in chapter 4, to represent when and how they were first introduced, how the focus in research shifted towards automation of existing solutions and take a look at future trends.

2. TERMINOLOGY

We will discuss certain terms in order to give them a uniform meaning in this research and to aid in understanding the following classification of test oracles. Our definitions are based on the widely-used ISO standard, the current version being ISO 2022 at time of writing.

SWC Seminar 2022 RWTH Aachen University, Germany.

- **Programs** are defined as code built on a specific programming language with the goal of solving specific tasks.
- The faults lying in the internal logic of the code are the cause of **errors**, which are defined as a difference between a theoretical condition and its real value.
- Each program produces a set of **results** derived from the processes defined by it.
- **Testing** a block of code consists of setting an expectation for the results of executing it and comparing it with the returned values.
- Programs being tested are considered **Software Under Test**, or SUT.
- One set of conditions and checks applied to a single program is considered a **test case**.
- **Test data** is data which is created or selected to be suitable for executing one or multiple test cases.
- The **test oracle problem** describes the challenge to determine if a test has passed or failed for a given set of test inputs.

3. CLASSIFICATION OF TEST ORACLES

Simple test oracles like input-output comparisons that explicitly check for equality between given and received information are common in practice due to their ease of implementation [13]. However, they are rarely discussed in scientific literature, as they do not hold possibilities for optimization or expansion because of this simplicity.
More complex test oracles can be divided into three main categories: specified, derived and implicit, each of which has multiple sub-classes that we will focus on in the following chapters [1].
Specified test oracles are designed according to formal specifications which are based on the desired behaviour of a system [1]. In our analysis we focus on two types of specified oracles called specification-language and assertions.
Derived test oracles evaluate the behaviour of a system based on information extracted from sources like documentation, system executions or previous versions [1]. In this chapter we look at the classes of pseudo-oracles, metamorphic testing and regression testing.
Implicit test oracles assess a systems behaviour by analysing general and implicit information like program crashes. Since these oracles are independent of any formal specifications or knowledge about the system they can be used universally for all programs [1].
The aforementioned classes are a vague categorization of test oracles, therefore for our analysis we split them into more specific sub-classes discussed in detail in the following chapters.

3.1 Pseudo Oracles

A program for which there is no oracle or for which an oracle cannot practically be implemented is called *non-testable*. There are three types of non-testable programs. Firstly programs where for a given input the output is unknown and the purpose of the program is to calculate the right output. Secondly are programs whose quantity of output is too large to verify. Lastly are systems impaired by tester's misconceptions about, for example specifications [3].
It is believed that many systems fall under the category of being non-testable, which brings up the question of how to verify the correctness of such programs [15].
One way to test a non-testable program is a *pseudo-oracle*. The approach of this method is to implement multiple versions of the same program in parallel by disjoint teams of programmers based on the same data and specifications. After completion the different programs are compared and if the outputs are equal the test is valid. Otherwise both programs are revised and examined again until all differences are removed. The comparison is taken over by monitoring both programs and can be automated to check larger amounts of test data in shorter amounts of time. Pseudo-oracles also solve the problems of non-testable programs type three, listed in the beginning of this chapter, since it is highly unlikely that both programming teams are subject to the same misconception [3].
To make practical use of pseudo-oracles some requirements have to be met. First independence of the pseudo-oracle is crucial to reduce the risk of misconceptions influencing the oracle and its original program. Second a convenient and high-level programming language to speed up the development of the pseudo-oracle, so that programming and debugging is not overly time consuming. Third the original program will be used frequently in order to compensate the extra time and work of producing multiple versions of the same program. Fourth specifications need to be complete, precise and equal for each individual team to minimize the differences between their results. Last is the suitability of test data, for example programs which output a boolean value of true and false are more difficult to compare for correctness then programs with an integer output. This is the case since it is highly unlikely for two programs to produce the exact same integer value if one program is not correct. When comparing two boolean values the confirmation is less meaningful since the chance of accidentally hitting the right answer is higher[3].
A second method can be applied for code belonging to category 2 that has too much output to verify. For checking such programs it can be useful to simplify the data and run tests on these reduced inputs and outputs. However, it is clear that results from simplified tests can not be taken as a measurement for the entire program, since often the complex test case invoke errors and are overlooked by this method. Therefore it can only be of assistance with finding specific types of errors. Another more intuitive method is to accept reasonable results which, however, can not be fully validated as being correct. This can be done by narrowing down the scope of possible correct outputs towards an acceptable interval or by calculating the probability of a result to be right [15].
Testing without a test oracle can mainly result in two consequences that we want to discuss now. In order to do so we distinguish two different results that are most likely to happen. First is a correct output but the test's verification fails and labels the output as incorrect. Although less likely and often caused by a tester's mistake, this always results in additional time and labour cost for someone to find the reason for the false error. In a worst case scenario the tester also changes the code in order to fix the problem and therefore adding new real errors [15].

Second, and more regular, is the consequence of a test output being wrong but the test result verifies it as correct. It is rarely the case that programs, which were subject to tests, do not include any errors, but it should be possible to rely on the fact that the sections of an application that were previously tested are truly correct [15].
In order to support and simplify testing of non-testable systems Weyuker suggests the following list of information that should be included in any program's documentation as a guideline for future testing [15]:

1. Specify the testing criteria which the test data is based on
2. The amount of testing criteria fulfilled in relation to unfulfilled criteria
3. The actual data used for testing
4. The output for each individual input chosen from the test data
5. Give a description of how outputs were verified as correct or incorrect

This concept can also be applied to and be beneficial for the following classes of test oracles.

3.2 Metamorphic Testing

Metamorphic testing describes the approach of taking existing tests and test data and transforming them, commonly used if the test data is complex and time consuming to compute in execution since there are no values beforehand and therefore need to be generated during runtime. The idea or goal of metamorphic testing therefore is that the new tests thus generated should then calculate the output faster and more efficiently compared to the original tests. As an example take an input x and the function f which computes the output of the program *f(x)*. The goal now is to find a metamorphic relation so that for any given input *x'* its output *f(x')* can be predicted using *f(x)* [12].
Testing with metamorphic relations is time- and labour-intensive for even small tests. Especially the input conversion requires a lot of computation and can be extremely complicated if the given data is not in a format that is legible for humans. In addition the output verification faces difficulties if, for example, the application allows multiple correct results for a single input [12].
Test oracles based on metamorphic testing still face problems limiting their effective usage in practice. The most prevalent challenge is reducing the time and costs necessary to modify each input while increasing its accuracy and correctness. Another obstacle is the comparison of an output to its computed relation as it also struggles with increasing data size and is susceptible to errors [12]. These problems might be solved by automating the process but more on that in chapter 3.6.
One example to illustrate metamorphic testing is finding a shortest path in an undirected graph. Take two nodes x and y in a graph G. The goal now is to verify if the computed shortest path *G(x,y)* is correct. The simplest metamorphic relation would be to test if the same path backwards, so *G(y,x)*, results in the same path length [2].
In order to connect this chapter with section 3.1, it can be said that metamorphic testing approaches are also a good method to test non-testable programs. Take the first class of non-testable programs where for a given input the correct output is unknown. Tests can still be carried out since metamorphic testing does not require any detailed knowledge about the program, because if an output for a given input is not as expected errors can still be detected [12].

3.3 Regression Testing

Regression testing describes the concept of comparing a new version of a program to its predecessor to check if it is still performing according to its specifications and requirements. It is based on the assumption that the previous version of a program can be used as a test oracle for the current specifications [1].
A distinction is made between *adaptive or perfective maintenance* and *corrective maintenance*. The former describes the case that the specifications change from one program version to the next, in which case all affected tests need to be updated or reconstructed in order to correctly represent the new specifications. In comparison, the latter describes the opposite case, so that the specifications did not change and the test does not need to be updated or reconstructed [14].
There are two ways to compare two versions of a program. On the one hand, the *retest-all* strategy, as the name implies testing the correctness of everything but therefore devoting more resources and time, which is not worthwhile for smaller changes. And on the other hand, the *selective* strategy only testing a partial amount of the tests generated for the previous version and therefore minimizing resources spent. The second method has the goal of only needing a new test run if something in the section of code it covers changed, which further reduces costs and time spent for testing. Though it needs to be mentioned that in comparison to the goal mentioned above, it also adds a new problem of finding dependencies in the code that effect the block of code covered by the selective test oracle [14].
Take an object-oriented language like Java as an example. A *selective* strategy may find the affected code sections in a class but will miss all other affected parts which are dependent on the changed class. In order to also cover these indirectly affected classes new methods like collecting coverage information need to be added in order for *selective* strategies to be correct. Therefore these methods also collect data about all affected classes for each individual test, which then can be used for the next version of the program to select all tests that need to be rerun to verify the correctness of the updated version of a system [8].
According to Wahl a testing process with a selective strategy can be described in six steps [14].

1. Analyse the differences between both system versions and the code sections affected by the changes
2. Choose the tests related to the affected code sections
3. Test the affected code section on the basis of its related tests
4. Check the test results for correctness or identify failures
5. If an error is found then localize and correct it
6. Update the new program and its tests

The concept of regression testing described above applies to three different levels of a program - the unit, integration and system level. At each level different failures can occur. Regression testing in general applies to all levels combined but there are also specific test oracles for each individual one [14].
The unit level is commonly focused on by regression testing methods and its goal is to verify correct behaviour according to its specifications, for each individual software module [14].
At the integration level regression testing puts a focus on gradually merging the individual software modules back together to form the original program. In this process errors can be found that did not occur before at the unit level testing. This procedure is of high importance since around 40% of failures are caused by incorrect merging of individual modules [14].
On the system level regression testing checks if the complete program functions according to its specifications. Therefore it first has to check if there were any failures on the integration level and that the merged modules meet their requirements and functionality. Thus this testing procedure does not require any information about the contents of each individual model [14].
When comparing different regression testing approaches four different characteristics are mainly considered. First is *inclusiveness*, meaning the range of tests chosen by the method to find failures triggered by changes in the code. Second is *precision*, representing the accuracy of only choosing tests that are relevant to revealing an error introduced as an effect of changes. Third is *efficiency* as a measurement of costs generated by the computation of a testing process. And as a last criterion there is *generality*. As the name implies, it is describing the adaptability of a test oracle to be used for different languages and applications. When looking for a test oracle these categories can be used to make a decision, for example if a program is required to be extremely reliable, a safer method with high inclusiveness is best to choose in order to guarantee finding as much errors as possible. On the other hand, if the goal is to minimize the time a test oracle needs to verify the correctness of an application, a version with high precision is the best choice since it reduces the amount of tests needed for confirmation [14].
To come back to the beginning of the chapter we now take a look at the juxtaposition of cost effectiveness regarding *retest-all* and *selective* testing strategies. In most research the focus is on *selective* methods since it is presumed that these approaches incur less costs by only working with a fragment of all test cases. But as described above these methods include a selection process in order to choose the right tests to work with. Therefore if the task of inspecting and filtering test cases involves too high costs or can not reduce the number of test by a significant amount, *selective* strategies are equally or even less cost efficient in comparison to *retest-all* approaches. So when comparing the costs of these two methods it can be said that *selective* strategies are more efficient if the added expense of test selection is lower then the costs of running and verifying the excluded tests which are included in *retest-all* techniques [14] .

3.4 Specification Languages

Test oracles based on specifications are designed to describe the desired behaviour of a system in a formal manner. Once a specification-based oracle is defined it can be reused for example in regression testing. Therefore the cost of creating the oracle only has to be raised once and pays off by its reuse [13].
In the following we will describe the concept of a specification-based test oracle on the basis of an elevator example given by Richardson et. al.. The system, in this case the elevator, has to respond to specific inputs. For example a button to call the elevator to a certain floor or the buttons inside the elevator to move to a certain floor. Furthermore it is restricted in its actions. For instance it should not move when the doors are open. In order to formally represent these requirements specification languages are used, examples are formal languages or state charts. For the example we look at a formal language to represent the events and temporal properties of an elevator. Actions are represented by terms of the language like **open(elevator)**, **close(elevator)** or **set_direction(elevator, direction)**. After all events are defined the properties of the system can be described. An elevator should not move with open doors therefore a restriction is made which is a chronological sequence of events that must be fulfilled in order for the system to behave correctly. This could be represented as follows: **set_direction(elevator, direction)** $\Rightarrow$ **(up** $\vee$ **down)** $\Rightarrow$ **open(elevator)** so the elevator only opens after it has moved up or down. Now at each control point, for example when the elevator reaches a floor, an assertion is made to check if the system complies with its specifications [13].
Specification languages like the one described above model all the states and operations of a system and therefore are called *state-based specifications* [1]. Another subclass is called *state transition systems* and, as the name implies, the properties of a system are depicted as a graph displaying the transitions between each state. The output of a system is either the final state or the set of transitions the system went trough. Specification languages only approximate the system so differences in its behaviour will therefore occur at some point and the test oracle needs to be revised in order to solve the conflict. As a result it is still an open problem to find models that exactly represent its system which depends on if the system's behaviour is observable and deterministic [1].
In general there are still three challenges for specified test oracles to be solved. First is, as briefly touched on before, the task of finding formal specifications that are abstract enough to work with. This problem leads to challenge number two, the inaccuracy of a specified test oracle caused by non-realisable properties or incomplete coverage of all the required specifications. Lastly it remains a challenge to find an accurate method for comparing the test oracle output to the program output. This problem arises because the results of a specified test oracle are on a higher abstraction level then the results of the actual program. In addition the output of an oracle might only be computed to a limited extent or is simplified in comparison to the actual output [1].

3.5 Assertions

Assertions as a test oracle are not separated from the original program but are integrated into the code and executed at runtime. They are implemented as boolean expressions where a value of **true** symbolises correct execution and a value of **false** indicates an error [1].

In general assertions are also classified as specification languages but differ from them in two ways. Firstly, the fact that assertions can access and use the programs variables so the probability of a wrong representation in the specification language is reduced drastically. Secondly, assertions are not written after the SUT is finished but alongside its development [1].
In practice a classic assertion test oracle has constraints to the amount of properties it can check at each point in the program [1]. Therefore all the classes analysed above extend the principle of assertions and add new methods and functionality in order to create better and more efficient test oracles.

3.6 Machine Learning for Test Oracles

Machine learning does not introduce a new solution to the test oracle problem but aims for automating already existing solutions. The goal is to aid in or even completely take over the creation of test cases and also executing the task of a test oracle to verify the correctness of a program. In research the main focus is on the second target of automating the generation of test oracles. Since current methods are often individual for each program and involve a considerable amount of human resources the automation of this process would result in massive cost and time reductions. However, there are currently still limitations in the process of creating an oracle due to the human involvement in finding good quality test data in high quantities and then training the machine learning algorithm [5].
Many developed methods and especially earlier approaches are reduced to a smaller set of outputs as they struggle with multiple and more complex outputs [5]. Though machine learning techniques brought forth recently make use of more complex and larger-scaled neural-networks for example and therefore are able to work with more complex problems and find solutions for them [7], [4], [6].
In general machine learning methods make choices based on the analysis of the given data and its structure for patterns and correlations. Automation approaches mainly focus on three types of test oracles, test verdicts, expected output and metamorphic testing. Test verdicts try to predict the results of a test for a specific input. Expected output approaches attempt to predict the actions of a system caused by a certain input. The results can vary from exact outputs to a wider more abstract set of outputs. The latter makes it difficult to forecast a system's behaviour if it is more complex. Metamorphic testing was already discussed before, here machine learning approaches try to find metamorphic relations based on the data it is given of a new program [5].
Test verdict methods take over the task of creating the information needed to verify the correctness of a program and then also perform the verification process. In comparison the other two approaches assist in or completely take over the definition of oracle information and specification but do not take part in the confirmation of correctness, since they only provide the information needed for that process [5].
However, there are still open problems which need to be solved in order to include test oracles created by machine learning algorithms into practical use. The first challenge is to find adequate training data since it needs to have the necessary contents and its size can often reach enormous amounts. In addition it still requires a significant portion of human involvement in order to produce these training data sets and can be highly time consuming since the program needs to be run numerous times to collect the necessary data to even start the training process [5].
Considering the class of metamorphic test oracles, most approaches are not yet ready to be used in practice since they can not overcome the need for human involvement. In order for a machine learning algorithm to learn which metamorphic relations hold or not a person with expertise has to manually add labels to the code to mark existing relations. This results in high costs before the actual training process has even started [5].
Once the learning algorithm has produced a test oracle it is fixed and does not change during its use. This can cause inaccuracy if the training data was not complete, therefore adding the necessity of retraining the oracle with further data if results are not correct or when new specifications are included in the program [5].
In general there are still three open problems in research yet to be solved. First the readability and usability of outputs which machine learning test oracles give, since they can differ from current test oracle approaches. Second the correctness of the aforementioned methods because it is believed that their results are at most partially correct. Third is the acceptability of users, meaning the aim to find a good way of organizing the cooperation of human involvement and expertise with automation of creating, executing and assessing tests [10].

3.7 Implicit Test Oracles

As described in the beginning of chapter 3, implicit test oracles are based on finding and analysing more indirect and universal information of a system to verify its correctness [1]. An often used example is buffer overflow errors, since they almost always indicate an incorrectness of a system. Considering these types of information an implicit test oracle uses, they are applicable to almost all programs and do not require knowledge about the system and its specifications. Also these approaches are often not standalone solutions for verification but are built on already existing processes which detect those obvious errors, like system crashes. Therefore an implicit test oracle defines correlations between a certain error as input and an reaction as output. However, it should be noted that these approaches are not always guaranteed to be correct for each program since in some cases even system crashes can be a desired outcome [1].
One example we want to look at is called *fuzzing*. The approach of fuzzing describes a method to detect implicit errors like system crashes in a time and cost efficient manner. It is based on the idea of generating random, or where the name comes from, *fuzz* inputs, and then testing the system with that created test data. This method works since it only analyses implicit information of the software under test which should result in the same output for each applicable input. If, however, an error is found in the testing process a report is made containing the specific error and the input that caused the problem. A common application for *fuzzing* is in the area of software security to find system vulnerabilities caused by a buffer overflow, a leak in the memory or exceptions that are not being handled [1].

4. TEMPORAL EVOLUTION

The first use of the term "test oracle" can be found in a 1978 paper by W. E. Howden [9]. Howden defines test

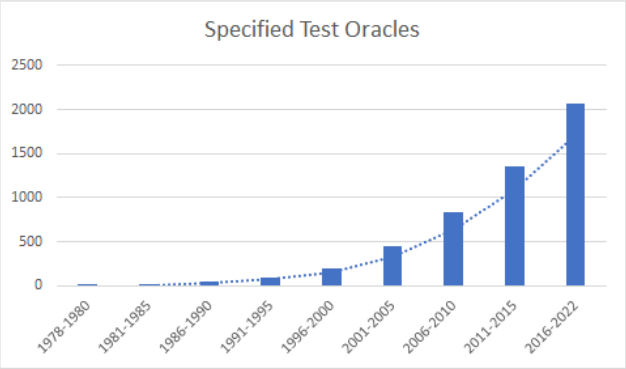

Figure 1: Results of the search query "test oracle" + ("assertion" OR "specified oracle" OR "specification language")

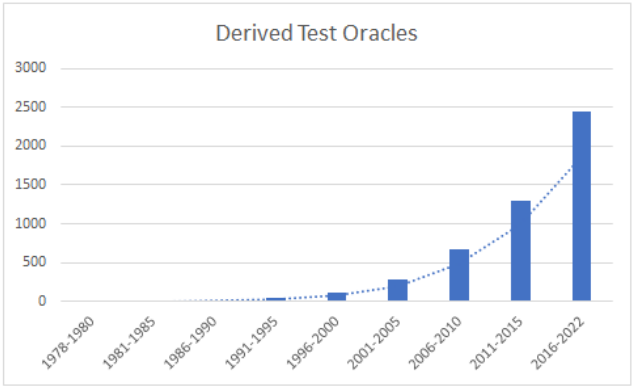

Figure 2: Results of the search query "test oracle" + ("derived oracle" OR "pseudo oracle" OR "metamorphic testing" OR "regression testing")

oracles as an assumed to exist tool to check correctness of an output. Notably he also differentiates between multiple types of oracles, namely formally and informally defined, though suggests no further categorization. His work is not the first to focus on the testing problem, as papers from as early as the 1950s already looked at other techniques, e.g. finite state machine testing, but it remains one of the most influential particularly because of the now staple term.
Following Howden's proposed terminology, techniques were being developed increasingly often, with Barr et al. observing that the 1990s spawned the most out of any decade [1]. The cumulative amount of publications also increased every year up to 2012. Replicating the Google Scholar query method of Barr et al., we observe that this trend continues into the 2020s, suggesting that the field continues to be receive interest by researchers.

In order to get a brief overview of how research trends developed over time we run four different web searches on *Google Scholar*, one for each group of test oracles, namely specified, derived and implicit oracles as well as one search for automation of oracles via machine learning. The queries consisted of the keywords *"test oracle"* followed by "*test oracle class*" or "*test oracle sub-class*" for all classes discussed in the previous chapters resulting in the following four query types:

1. "test oracle" + ("assertion" OR "specified oracle" OR "specification language")
2. "test oracle" + ("derived oracle" OR "pseudo oracle" OR "metamorphic testing" OR "regression testing")
3. "test oracle" + ("implicit oracle" OR "Fuzzing")
4. "test oracle" + ("automating test oracles" OR "automated oracles" OR "machine learning" OR "deep learning")

The results of each individual query were added up for each class and presented in figure 1, 2, 3 and 4.
This approach on analyzing the amount of publications involves some minor drawbacks and inaccuracies. First of all *Google Scholar* does not cover all publications and therefore might miss papers which were only published in specific publications or academic journals. Furthermore the papers found might not exactly fit to the desired topic, but these cases are minimized by requiring that the keywords searched for need to be exactly contained in each query result. From this it can be concluded that our approach might not be 100% accurate, but is precise enough to give an overview on how research trends and interest has developed over time for each class of test oracles. In the following we will first analyse each category of test oracles on its own and then give a comparison between all four.
To start off we take a look at the first search query of *specified test oracles* displayed in figure 1. It can be seen that since the term *test oracle* was first introduced in 1978 research took a slow start of 4 to 10 published papers per 5 year period until 1990. From 1991-1995 onward research in the field kept growing up to a steady pace of around one to three hundred new papers published in each time period until 2010. From then the interest in research on specified test oracles grew rapidly reaching 713 new papers between 2016 and 2022. The amount of published papers in this field reaches an amount of nearly 2100 papers by the year of 2022. This leads to the conclusion that specified test oracles are up to this day a research problem of high interest and this trend is likely to continue in the next years, growing even faster than in the last time periods.

Next we analyse the results of the next search query on *derived test oracles* presented in figure 2. Again it can be said that after the term *test oracle* was first coined in 1978, research in derived oracles started off slowly with the first papers being published between 1981 and 1985. In the first time periods from 1986-1990 until 1996-2000 between 12 to 80 papers were published per time span starting off a bit slower in comparison to research on specified test oracles. But from 2001 onward research expanded quickly nearly doubling from one time period to the next reaching an amount of 1150 new papers between 2016 and 2022 and a total amount of nearly 2500 papers. Since publishing numbers keep rising it can be assumed that this field of research will continue to expand in the coming years.
For the third category of test oracles we take a look at the development in research of *implicit test oracles* shown in figure 3. Up to 2005 there was only one paper published which was between 1986 to 1990 leading to the conclusion that before this year the field of implicit test oracles did not arouse any interest for further research. However, from that point

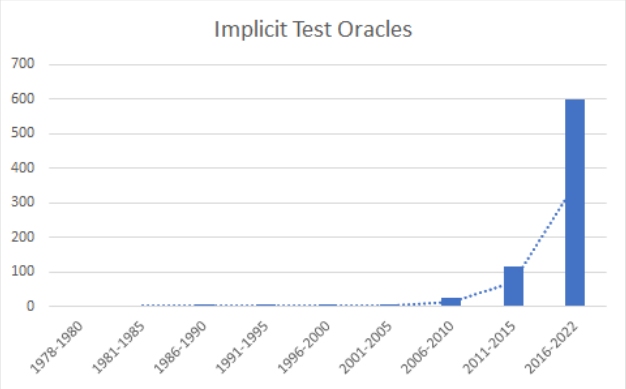

Figure 3: Results of the search query "test oracle" + ("implicit oracle" OR "Fuzzing")

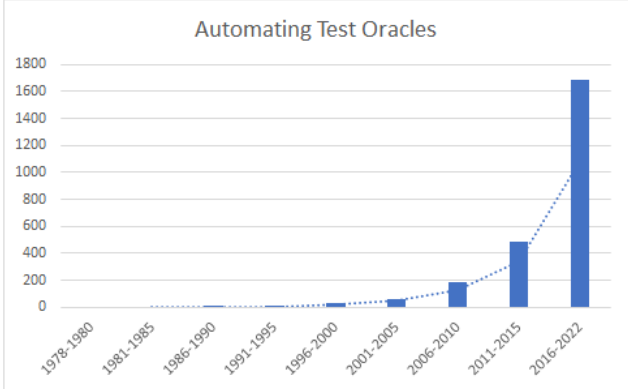

Figure 4: Results of the search query "test oracle" + ("automating test oracles" OR "automated oracles" OR "machine learning" OR "deep learning")

onward researchers started to investigate the topic more intensely, publishing 22 papers between 2006 and 2010. In the next time period the amount of publications more then quadrupled to 92 new papers and then increased five-fold in the next time period with 483. In total the research in implicit test oracles reached an amount of nearly 600 papers by 2022. Since that breaking point in 2010 the numbers are increasing rapidly up to today, leading to the conclusion that the focus on research in this field is starting to attract a lot of new scientists and that this class will be further investigated in the future.

At last we also want to take a look at the development of research in automating the creation and execution of test oracles with the help of machine learning. The results of the query are presented in figure 4. Surprisingly the first paper was already published between 1986 and 1990. From then research interest started to grow slowly with 12 to 28 papers per time period until the year 2005. After that the amount of publications increased rapidly more then doubling every five years until 2015 reaching 300 new papers from 2011 to 2015. Since then the field of automating test oracles expanded vastly, quadrupling by the end of 2015 to 2022 by 1200 new papers. In 2022 a total amount of roughly 1700 was reached. When looking at figure 4 it can clearly be seen how the interest in this approach is starting to attract many new researchers as it seems to promise great possibilities to increase time and cost efficiency in testing as well as decreasing the need for human involvement.

When comparing all four charts it becomes apparent that the temporal evolution of specified and derived oracles is quite similar even in the amounts of publications and only differ minimally. The same can be said for implicit test oracles and automation, their research trend curves are similar as well but they differ in their total amount of publications. Namely 600 for implicit and 1700 for automating test oracles. When ranking all four categories by their total amount of papers by 2022, *derived test oracles* are in first place, followed by *specified test oracles* and then *automation* of test oracles each by a gap of around 400 papers. Only *implicit* test oracles are far behind with a difference of more then thousand papers to the next class.

After analysing all four categories it can be concluded that research in all fields is still thriving and will likely continue this trend in the upcoming years. Although it must be said that the focus of research has shifted more towards improving existing solutions and also automating them instead of finding new solutions to the test oracle problem [1], [5].

5. CONCLUSION

In this paper we analysed different classes of test oracles and displayed their temporal evolution in research. To answer the first research question we divided test oracles approaches into three broad categories, specified, derived and implicit test oracles. The first category describes test oracles which are designed on formal specifications of a system in order to model the desired behaviour of a program. In our analysis we focused on two types of specified oracles called specification languages and assertions. The former implement a language in order to represent the properties of a system and creating conditions based on this language to check if the system acts according to its specifications. Assertions are boolean expressions integrated into the source code of a program in order to validate the correctness of a program at specific checkpoints.

For the next category of derived test oracles we analysed three different classes, pseudo-oracles, metamorphic testing and regression testing. Derived test oracles in general describe the concept of evaluating a systems behaviour based on information like documentation, system executions or other versions of the program. Pseudo-oracles are designed to test and validate programs for which no test-oracle exists since they are classified as non-testable. In order to achieve that the original system is programmed multiple times from multiple disjoint programming teams and their output is then compared. If they produce the same result the test is considered to be successful. Metamorphic testing describes the approach of transforming the current tests and test data into a new format, which is less complex and faster to compute while still being able to judge the original programs correctness based on the conversion.

As a last approach in the category of derived test oracles we looked at regression testing. This concept describes the process of comparing two versions of a program and ensuring that correctness still holds for the newer version, so that the changes in the program did not result in any errors or changes in behaviour.

For the last category of implicit test oracles we focused on one exemplary method called *fuzzing*. The idea behind *fuzzing* is to test the system on randomly generated inputs since the errors covered by implicit testing techniques should be the same for all input output pairs.
In addition to the three classes we looked at the automation of these existing test oracle approaches with the use of machine learning. Automation aims for either taking care of the creation of test data or executing the tests and verifying them and in some cases even do both jobs together. The focus, however, is mainly on the second task of creating a test oracle.
For our second research question we analysed a temporal evolution of the research on these topics. With this goal in mind we made four different search queries on *Google Scholar*, one each for specified, derived and implicit test oracles as well as automation. We checked the results in five year periods starting in 1980 up until 2022. All four queries resulted in a slow increase of research from the 1980 onward to around 1990-2000 depending on the oracle type. From this point on all four groups experienced a rapid increase in interest and therefore also in research and publications.
As a result of our analysis, we can state that the test oracle problem and its subfields are still of great interest for researchers but their focus has shifted more towards optimizing already existing solutions and automating them.

6. REFERENCES

[1] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[2] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: A new approach for generating next test cases. 2002.

[3] M. D. Davis and E. J. Weyuker. Pseudo-oracles for non-testable programs. In T. Shelter, S. Abraham, E. Friedman, and B. Levy, editors, *Proceedings of the ACM '81 conference on - ACM 81*, pages 254–257, New York, New York, USA, 1981. ACM Press.

[4] V. A. de Santiago Júnior. A method and experiment to evaluate deep neural networks as test oracles for scientific software. In *2022 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 40–51, 2022.

[5] A. Fontes and G. Gay. Using machine learning to generate test oracles: a systematic literature review. In G. Jahangirova and V. Terragni, editors, *Proceedings of the 1st International Workshop on Test Oracles*, pages 1–10, New York, NY, USA, 2021. ACM.

[6] C. Geethal. Training automated test oracles to identify semantic bugs. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1051–1055. IEEE, 2021.

[7] L. Grandinetti, S. L. Mirtaheri, and R. Shahbazian, editors. *High-Performance Computing and Big Data Analysis*. Communications in Computer and Information Science. Springer International Publishing, Cham, 2019.

[8] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. *ACM SIGPLAN Notices*, 36(11):312–326, 2001.

[9] W. E. Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, SE-4(4):293–298, 1978.

[10] W. B. Langdon, S. Yoo, and M. Harman. Inferring automatic test oracles. In *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*, pages 5–6. IEEE, 2017.

[11] L. J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, 1990.

[12] C. Murphy, K. Shen, and G. Kaiser. Automatic system testing of programs without test oracles. In G. Rothermel and L. Dillon, editors, *ISSTA 2009*, page 189, New York, NY, 2009. Association for Computing Machinery.

[13] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In T. Montgomery, editor, *International Conference on Software Engineering*, pages 105–118, New York, 1992. Association for Computing Machinery.

[14] N. J. Wahl. An overview of regression testing. *ACM SIGSOFT Software Engineering Notes*, 24(1):69–73, 1999.

[15] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.