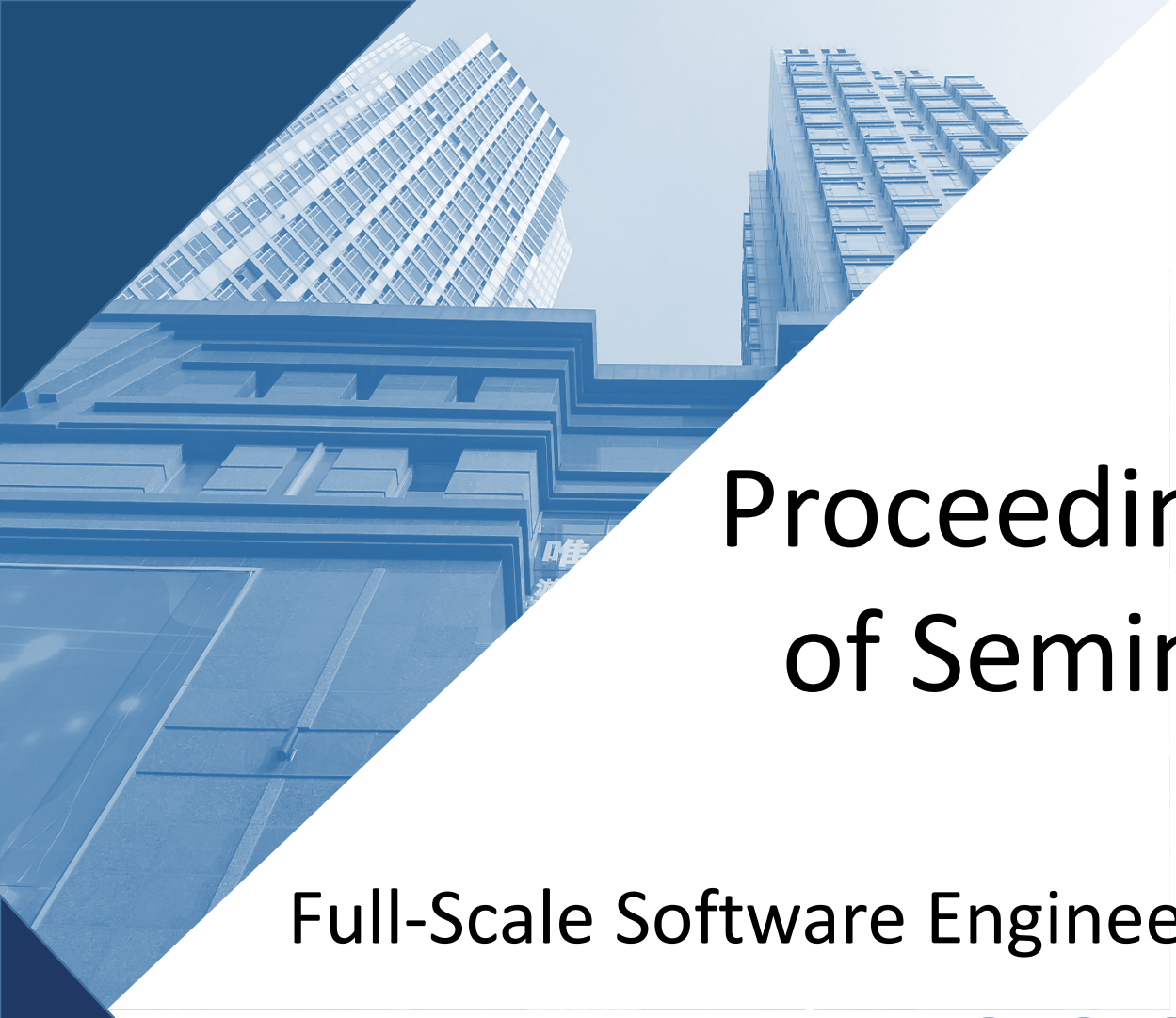


Proceedings of Seminar

Full-Scale Software Engineering

2025

Editors: Horst Lichter
Alex Sabau
Ada Slupczynski
Selin Aydin
Nils Wild

SWC Software Construction | RWTH AACHEN UNIVERSITY

Table of Contents

Prompt Engineering: Analyzing Developer Needs and Challenges

Atharva Jadhav
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
atharva.jadhav@rwth-aachen.de

Bora Avcu
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
bora.avcu@rwth-aachen.de

ABSTRACT

Prompt engineering is a critical technique for optimizing the performance of large language models (LLMs) which are revolutionizing problem-solving across domains. While existing tools and frameworks provide partial support for prompt creation, they often lack comprehensive coverage of the entire prompt development lifecycle. This gap creates inefficiencies and limits the ability of developers to streamline workflows and achieve optimal outcomes.

In this study, we conduct a systematic literature review to understand how developers engineer prompts and evaluate the effectiveness of current tools. We formalize the prompt development lifecycle, map existing tools to key activities and identify significant gaps that hinder their usability and effectiveness.

Our research includes an in-depth analysis of existing tools, focusing on iterative refinement, collaborative workflows and debugging features. Our findings reveal that current tools fail to fully support these essential aspects of the prompt development process.

To address these gaps, we propose actionable recommendations including the development of IDE-like environments, standardized libraries and collaborative platforms. These insights not only highlight the limitations of current tools but also provide a roadmap for future research and development to enhance prompt engineering workflows and improve developer productivity.

Keywords

Prompt Engineering, Prompt Engineering Tools, Prompt Development Life Cycle

1. INTRODUCTION

LLMs are revolutionizing the field of problem-solving[3] by producing output based on vast amounts of trained data and learned knowledge[27]. Models like GPT3[5] have integrated themselves in various fields ranging from education to software development, making it viable for a diverse audience. These models rely on natural language inputs, which guide their reasoning processes to generate relevant results[12, 21]. The quality of these outputs depend on two critical factors, such as the structure[12] and clarity of the prompts provided and the models' internal parameters, like weights. Consequently, understanding how developers craft these prompts is vital[17]. Prompt engineering, an emerging area of research, focuses on the creation, refinement and optimization of prompts to achieve specific outcomes[12, 21]. While various tools have been developed to aid in this process, they often fail to provide comprehensive support across all stages of the prompt engineering life-cycle. This study aims to investigate the patterns and activities developers undertake while designing prompts, evaluate the tools available to support them and assess their effectiveness in addressing developers needs while identifying limitations in prompt engineering. Based on these objectives, two research questions (RQs) were formulated:

- **RQ1**: Which activities are performed by prompt developers?
- **RQ2**: How are current tools supporting the defined activities?

To address these questions, the paper is structured as follows: Section 2 outlines the methodology, including the systematic review process. Section 3 presents the derived results, discussing the definition of prompt engineering, roles in prompt engineering, developer activities and tool evaluations. Section 4 discusses the implications of these findings and lastly Section 5 concludes the study by summarizing the key aspects and proposing suggestions for future research in the area of prompt engineering.

2. METHODOLOGY

This section describes the systematic literature review conducted to explore the emerging field of prompt engineering. The review process was guided by Parsifal, a web tool for managing structured literature reviews [1]. With the aid of Parsifal, we systematically reviewed multiple databases, employing tailored prompts and assessment questions to identify relevant studies, followed by data collection, quantitative analysis and lastly, evaluation.

2.1 Systematic Review Process

The systematic review process was conducted in several stages, beginning with the definition of research objectives

[1] https://parsif.al/

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2018/19 RWTH Aachen University, Germany.

and research questions. The PICOC framework (Population, Intervention, Comparison, Outcome, Context) provided a structured approach for formulating research terms, ensuring the questions were well-supported by relevant literature [7]. This framework also promoted the creation of database queries and the refinement of searches using interchangeable keywords. To thoroughly explore the activities, needs, tools and challenges of prompt engineering, we developed a detailed review protocol with specific questions and criteria. Furthermore, a comprehensive literature search, applying inclusion and exclusion criteria, was conducted across multiple databases to identify meaningful studies. Relevant information was systematically collected and analyzed to derive insights, which are presented in this study. The following sections detail the search strategy, including methods, search engines, data extraction and analysis.

2.2 Search Strategy

A core aspect of our methodology was the procedure used to collect key references, applying various criteria to determine whether the identified sources are essential for our study.

2.2.1 Search Engine

Ensuring the correct selection of suitable search engines is crucial for identifying beneficial academic papers that provide comprehensive, reliable and credible sources. The following sources were used as foundation for this study: *ACM Digital Library, Arxiv, IEEE, Science@Direct* as well as manually enriching the corpus with papers from *Google Scholar* [2].

2.2.2 Search Strings

Search strings were used to query multiple databases, which are listed in the Search Engine section 2.2.1. The structure of a search string consists of *AND*, and *OR* which connect *keywords* to find desired results. Creating an effective search string is an important step in obtaining reliable and relevant sources. The following search strings were respectively used for each research question:

- **RQ1:**("prompt engineering" OR "prompt design" OR "prompt optimization" OR "prompt developer" OR "prompt programming" OR "prompt programmer" OR "prompt engineer") AND ("activities" OR "techniques" OR "methods" OR "strategies" OR "practices" OR "process" OR "life cycle")
- **RQ2:**("prompt engineering" OR "prompt design" OR "prompt optimization" OR "prompt programming") AND ("tools" OR "frameworks" OR "IDE" OR "development kit")

These two search strings formed the foundation for the research conducted.

2.2.3 Inclusion and Exclusion Criteria

Since this is a new area of research, we had to apply a couple criteria in order to extract the most relevant papers for this study. Inclusion and exclusion criteria assist in the search of papers by determining which papers to consider in our selection or which papers should be excluded[7]. The following inclusion criteria (IC) were chosen:

- **IC1** The paper addresses one or more research questions
- **IC2** The language is English
- **IC3** The full paper is accessible

As for the exclusion criteria (EC) these were chosen:

- **EC2** Paper dives too much into machine learning
- **EC2** Paper with insufficient data to back up findings, which lead to weak conclusions
- **EC2** Excluding gray literature, which is not based on scientific findings

Given the dynamic and rapidly growing nature of this research field, we implemented stricter criteria to ensure selection of only high quality and relevant sources for our cause. These criteria were designed to guide us in identifying and including only the most suitable papers for our study.

2.3 Data Extraction and Analysis

After selecting suitable studies, we conducted a comprehensive review of all remaining papers that passed the previous filtering steps to determine which were relevant to our research. Given the novelty of prompt engineering, as illustrated by the rapid growth of interest in this field from 2021 to 2024 in Figure 1, extensive effort was required to identify studies addressing our research questions. Before 2021, relatively few high-quality papers were available, further emphasizing the emerging nature of the field. This trend necessitated a thorough review of identified studies and the manual inclusion of additional papers that met our inclusion and exclusion criteria to ensure a robust dataset. Following the selection process, we proceeded to the data extraction and analysis phase using Parsifal. This step involved systematically gathering detailed information from each study, including their objectives, methodologies and key findings relevant to our research questions. To answer these questions, we identified and extracted activities performed within the field of prompt engineering. These activities were mapped to corresponding roles responsible for their execution, with artifacts identified as the byproducts of these activities. The extracted data was categorized into themes such as documentation practices, challenges, existing gaps, future directions and tool improvements in prompt engineering. Our extensive search across multiple databases initially identified 98 papers, as shown in Figure 2. After applying strict inclusion and exclusion criteria, 50 papers were retained for in-depth analysis, culminating in a final selection of 31 studies used in this research. This systematic approach allowed us to extract valuable insights, identify patterns and gaps and propose strategies to advance this field of research within the domain of prompt engineering.

3. RESULTS

Our review of activities and tools in prompt engineering reveals a rapidly evolving field with distinct challenges and opportunities. The findings underscore the need for tailored strategies to support developers in optimizing prompts

[2]https://dl.acm.org/;https://arxiv.org/; https://www.ieee.org/; https://www.sciencedirect.com/; https://scholar.google.de/

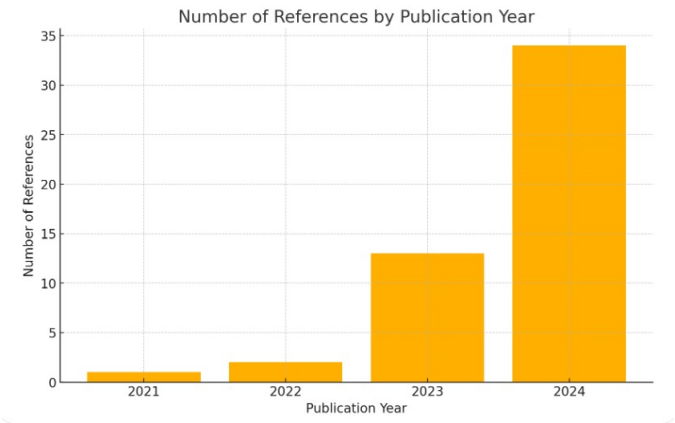


Figure 1: References distributed by year

Figure 2: Summary of the paper selection process

for LLMs. By examining the workflows and tools currently employed by prompt developers in prompt engineering, we identified best practices, gaps and emerging trends that currently define the field. This section is organized to address the two research questions RQ1 and RQ2 and provides insights into the activities performed by developers and the effectiveness of existing tools.

3.1 Prompt Engineering

Prompt Engineering is the process of crafting and iteratively refining input prompts to optimize the output generated by LLMs. This involves using natural language to create inputs that guide the LLMs towards solving a specific task desired by the user, which can range in complexity[21, 16]. Furthermore, its main capability is addressing problems and generating desired results according to the given task and giving an output in natural and understandable language. Prompt engineering serves as a tool to bridge the gap between a user's intent and an LLM's capabilities by systematically structuring prompts, employing techniques like zero-shot, few-shot, Chain-of-Thought(CoT) prompting and accounting for the models limitations[21, 22]. Part of this process is defining the problem in a problem frame, refining the prompt iteratively and lastly evaluating of the output. These steps are very similar to the way how LLMs are trained. Essentially prompt engineering is used to improve, refine and tune the output of LLMs[1, 12, 4].

3.2 Role of a Prompt Developer

A prompt developer is a specialist skilled in creating, refining and tuning prompts to interact effectively with LLMs by iteratively applying techniques to improve the prompts. This role requires a good understanding of the used models capability, limitations and behavior in regard to varying inputs. Prompt developers engage in various activities throughout the prompt development life cycle, each of which align with their role specific task. This role emphasizes tasks such as problem framing, LLM Selection, designing, executing and evaluating solutions. Furthermore, ensuring that activities are systematically structured to achieve optimal results. Engaging in these activities as a prompt developer results in artifacts, as by product of each specific activity in the development cycle [2]. Defining the role of a prompt developer provides a clearer understanding of their responsibilities and tasks, enabling us to more effectively address the previously defined research questions.

In addition, an important aspect of a developers work flow is documenting each step of the way in order to see what step has to be improved on the path to the desired output. The documentation may be in the form of a prompt repository, templates and general documentation of behavior, which benefits the developer in the long run to facilitate iterative development and reproducibility. The expertise of a prompt developer is essential in influencing the LLMs output for different use cases, ranging from generating creative content such as pictures, to solving complex computational problems in areas of math and coding[28, 9].

3.2.1 Roles in Prompt Engineering

Part of our research was to identify the various roles involved in prompt engineering and the terminology used to describe them. While most studies we examined lacked a precise definition for specific roles in prompt development, they suggest that anybody who can speak and is fluent in natural languages, primarily English since LLMs are trained on English data sets, can be (prompt-) programmers [21]. The following Table 1, summarizes our most notable findings:

Name	Paper	Frequency
Prompt Programmers	[12, 19, 15]	3
GenAI users/ Users	[23, 4, 6, 26, 14]	5
Researchers	[23, 24, 10]	3
Technologist	[23]	1
Programmers	[11]	1
Prompt Engineers	[3, 31, 30]	3
Developers	[10]	1

Table 1: Frequency of role names in reviewed papers

The terms "Researcher" and "Technologist" in this context refer to individuals who engage in activities such as prompt hacking to test and uncover vulnerabilities in certain tools [23]. Interestingly, most studies refereed to people interacting with prompts as "Users" as reflected in Table 1. Indicating, the lack of consensus on role definitions, which furthermore underlines the novelty of this field. Additionally, much of the research focuses on providing guidelines to help improve prompts, further emphasizing the accessibility of prompt engineering as a profession. Hereafter, we refer to the individual interacting with LLMs to create prompts as a prompt developer.

3.3 Prompt Development Life Cycle

The following section showcases the development cycle of a prompt, with each major activity highlighted with the corresponding artifacts generated through performing those activities. In order to obtain the desired output, Prompt developers often refine and test their prompts in an iterative

manner [25][8][17]. That is why we have to take a deeper look into the performed activities and workflow of a prompt developer.

3.3.1 Activities

The process of prompt engineering involves a series of iterative activities , which reach from framing the problem to documenting every step, that developers continuously perform in order to achieve the desired output. Below in Figure 3 we introduce this process, which highlights the performed activities throughout a prompt development life cycle in the form of a process diagram:

1. **Problem Framing**: The first activity developers typically undertake is framing the problem. In this step, they formalize the goal and define the scope of the task they aim to address. This involves making key decisions, such as determining the necessity of specific input data and acquiring the required information. Depending on the problem's nature, developers may structure the desired output to meet specific requirements[25, 28]. However, if the problem does not necessitate a particular output format, developers may allow the LLM flexibility in structuring the response, provided it fulfills the intended purpose[8]. Additionally, developers establish metrics to evaluate the effectiveness of the prompts, ensuring alignment with defined objectives[17, 14, 1, 9, 16].

2. **LLM Selection**: After framing the problem, developers proceed to select the most suitable LLM for their task[8]. This decision involves evaluating the capabilities of different models to assure alignment with the problems requirements[17]. In some cases, developers may experiment with multiple LLMs to compare their performance. Moreover, developers carefully consider the limitations of each model, such as token limits, domain-specific expertise or response accuracy, to make an informed choice[25].

3. **Prompt Design**: At this stage, developers select an appropriate prompting technique, such as standard prompting, zero-shot, few-shot prompting or CoT prompting based on the task requirements [22, 12, 21]. Using the insights gathered during problem framing, they craft a structured prompt that effectively integrates the relevant data. Developers often include additional explanatory text to clarify the task they want the LLM to perform on the input data[8]. Prompt templates are commonly employed to streamline the design process and ensure consistency[1, 17]. In some cases, developers may also include specific parameters within the prompt to adjust the models behavior or fine tune the output. The prompt design process is guided not only by the tasks need, but also by the limitations and capabilities of the chosen LLM[25, 28, 9, 29].

4. **Prompt Execution**: Once the prompt is designed, it is executed by providing it to the selected LLM. The execution process involves submitting the crafted prompt, often alongside additional contextual information or algorithmic hints, to obtain a solution from the model[28, 17]. Prompt execution is not just about running a static query, but about engaging in a conversational and adaptive process to guide the LLM toward producing a correct and optimal solution[8, 1, 9, 16, 19].

5. **Output Evaluation**: Once the model generates an output, developers evaluate its quality using predefined metrics established during the problem-framing phase[25]. This involves analyzing the output to ensure it aligns with the desired objectives and identifying any discrepancies or deviation from the desired output[17]. The evaluation process often highlights areas where the prompt or parameters need refinement, driving iterative improvements in the prompt development cycle[8, 28, 1, 21, 9, 16, 6].

6. **Documentation**: This activity involves recording the developed prompts, their generated outputs and the corresponding evaluations of those outputs[8]. It also includes recording successful final prompts descriptions, versions, the failures of the engineered prompts and also cross check with defined metrics. Documentation serves as a critical reference point for iterative improvements and knowledge transfer, enabling developers to refine prompt design and ensure reproducibility. Additionally, structured documentation facilitates the identification of effective prompt patterns, support debugging process and contributes to the better understanding of model behavior in various contexts [25, 17, 28, 1, 4, 26].

3.3.2 Decision Points

During the prompt engineering process, developers face several key decision points which influence the final outcome of the prompt. These are outlined below:

1. **Result Quality**: Developers assess whether the output meets their expectations and aligns with the desired outcome. If the result is satisfactory, they accept the solution, otherwise they proceed to refine the prompt for improvement. Which leads up to the next key decision point[8, 17].

2. **Refinement feasibility**: Developers evaluate whether the further refinement of the input prompt is feasible. Since refinement can be a time intensive aspect, developers may decide to discontinue this process if they think that it would not yield the desired improvements[8, 25, 17]. Else the developer jumps back to the Prompt Design Activity and starts remodeling their prompt with the gained knowledge as seen in 3.

   Prompt developers determine feasibility and quality by assessing whether adjustments can be made, such as:

   (a) Add or remove new words or update parameters in the existing prompt [8, 17]

   (b) Modify or explore new prompt structure or prompting techniques[25]

3.3.3 Artifacts

Artifacts are items that are produced during a development process, in our case in the prompt engineering development cycle. These can range from bug reports to general information about the process. Each activity in a software development process usually yields an artifact, which

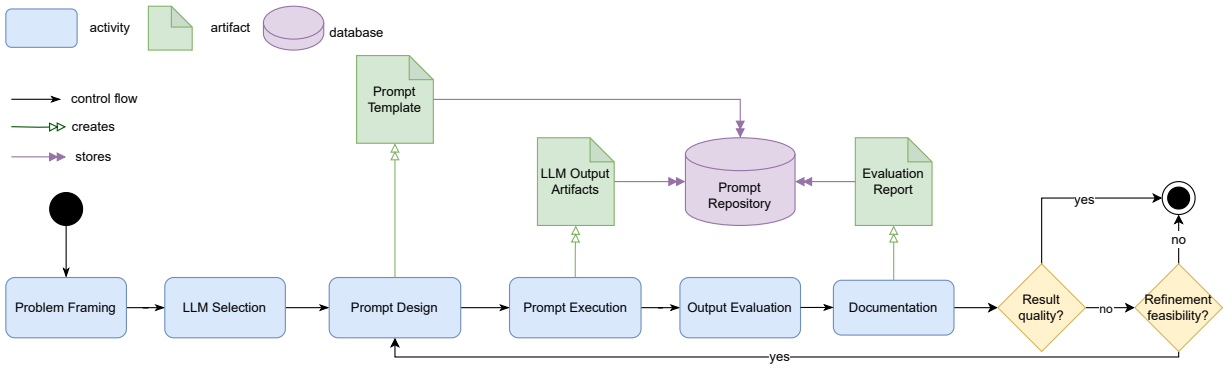


Figure 3: Prompt engineering development cycle

then can be used to document and improve said process[18]. Thus, the development cycle for prompts produces the following artifacts:

- **Prompt repository**: Since prompt engineering is an iterative process, multiple prompts are generated across iterations. These prompts can be systematically organized and maintained in a dedicated repository for easy access and future reference.
- **LLM Output artifacts**: Depending on the nature of the problems being addressed, developers often produce various output artifacts, capturing the results generated by the LLM during the prompt engineering process. These artifacts will also be saved in the prompt repository.
- **Prompt templates**: During the refinement process, developers may design specific prompt structures that consistently yield effective results. These structures can evolve into reusable templates for future tasks, enhancing efficiency and standardization.
- **Evaluation Report**: This artifact documents the analysis of outputs over multiple iterations, providing insights into performance, areas of improvement and the effectiveness of prompt adjustments.

3.4 Tool Evaluation

The input to LLMs being natural language simplifies the interaction between developers and the models. However, the iterative nature of this interaction presents several challenges. Structuring prompts effectively, understanding how to prompt specific LLMs and evaluating output quality are factors that can complicate the task of generating optimal prompts[12, 4, 20]. To address these challenges, several tools have been developed to aid in prompt generation. We evaluated these tools alongside the current state of art to assess their applications and features. The following sections offers a comprehensive overview of our findings.

3.4.1 Tools Overview

The central theme of prompt engineering tools revolves around the idea of iterative prompt development and its refinement. These tools consider factors around this theme and aim to simplify and accelerate prompt development. Since developers can work on specific tasks like code generation[8] or image generation as studied in[17], we focused on tools that are employed for use-cases namely text generation, image generation and code generation.

For text generation, we reviewed the Prompt-Deck[6] and PromptLayer[20] tools. PromptDeck employs a card-based system, where each card represents a specific task within a larger project. These cards are interconnected to create a cohesive workflow. Similarly, PromptLayer provides a comprehensive interface for managing the life cycle of prompt development. In addition to its feature-rich user interface, PromptLayer offers SDKs for streamlined development and supports integration with multiple LLMs.

GenLine & GenForm[13] and CoPrompt[11] were reviewed as code-generation use-cases. GenLine and GenForm are built to be tools that interact with generative macros. GenLine works on an in-line request while GenForm is a form that allows input description and generates code. CoPrompt revolves around the need of prompt co-engineering and is built for developers to work together.

Finally, Promptify[4] and PromptCharm[28] are tools that focus on image generation tasks. They provide a rich interface to develop images based on Stable Diffusion. Promptify allows clustering of images while PromptCharm provides additional support like direct inpainting of images.

3.4.2 Tool Features

Based on our analysis of developer activities and the tools they use, we have identified key features and gaps that support and enhance their efforts. The following section provides brief descriptions of these features and highlights their relevance to the development process, alongside the Figure 4 for easier cross comparison.

- **Problem Details**: Developers start the prompt development journey by framing the problem, as mentioned in the first activity. Thus, the Problem Details feature is essential to document these details, serving as a reference point throughout the development process and ensuring alignment with the defined goal[6].
- **Prompt Editor**:Designing a prompt involves structuring inputs and crafting a suitable text that aligns with the tasks requirements. A powerful prompt editor is essential for enabling developers to create, refine and iterate on prompts efficiently[6, 20, 13, 11]. Advanced capabilities such as improvement suggestions further improve productivity while prompt developing[4, 28].

- **LLM Selection**: The selection of the most appropriate LLM is a crucial step as seen in the life cycle, requiring tools to support integration with multiple LLMs[6, 20]. This feature allows developers to compare models, cross evaluate their capabilities and choose the best one suited for their objective, enhancing flexibility to switch between models, in order to improve outputs.
- **Iterative Refinement**: Iterative prompt refinement lies at the core of prompt engineering, enabling developers to continuously enhance their inputs prompt to achieve more accurate and optimized results throughout the development life cycle[4].
- **Version Control**: Given the iterative nature of the prompt development life-cycle, it is essential to persist artifacts such as prompts and their outputs. Version control plays a critical role in managing these artifacts, ensuring that both prompts and their corresponding outputs are accurately stored and tracked throughout the process. Developers can use the history to understand their changes and look for improvements while they develop prompts[11, 28].
- **Collaboration**: Prompt developers may need to co-engineer prompts to solve complex problems. Therefore, having a tool that enables collaboration during prompt development is essential. This capability aligns with the activities that developers undertake throughout the development process[11].
- **LLM Parameter Tuning**: Certain LLMs require parameter tuning to optimize their outputs alongside the input text. During prompt design and refinement, developers often adjust parameters to achieve better results. Environments with integrated parametrization capabilities can significantly assist developers by enabling faster and more efficient output optimization[20].
- **Evaluation**: Evaluating output is a crucial step in the prompt development life cycle as it decides necessity of further refinement. It is essential for tools to support evaluation processes, with criteria varying based on the use case. For instance, in image generation, developers need to visualize the output [17, 28], while in text generation, the output must be assessed against several criteria, as outlined in [6].
- **Custom Prompt Templating**: Prompt design and refinement often reveal recurring patterns in input prompts. Developers may wish to create templates from these patterns to reuse in future projects or iterations [17]. This feature facilitates developers to store and manage prompt templates efficiently [20].

While the reviewed tools provide essential features supporting various aspects of prompt engineering, they often fall short in addressing the entire development life cycle comprehensively. In particular, enhanced support for collaboration and robust documentation features could significantly improve developer workflows. On the other side, a critical activity like prompt refinement which is part of the entire development cycle is widely supported across most tools, providing essential assistance to developers in one of the most vital stages of the process.

4. DISCUSSION

The study highlights that prompt engineering is a rapid -ly evolving field driven by its ability to address a wide variety of problems with exceptional precision. The central 'define-refine-test' idea for prompting is widely recognized and serves as the foundation of effective prompt engineering. Our analysis of RQ1 reinforces this concept but also uncovers additional vital activities necessary for developers to achieve optimal results. Activities such as Problem Framing and LLM Selection are critical for setting the tone of prompt development while documentation emerges as a cornerstone for enabling collaboration, facilitating faster development and ensuring systematic refinement of prompts.

The analysis of tools based on different use cases demonstrates that the central idea of prompt engineering is well captured by all the tools. All tools have the features of prompt editing and allow refinement of these prompts based on results. These tools allow developers to initiate the prompt development process, however they do not fully unlock the potential of LLMs. Prompt engineering could be employed for large-scale or complex projects. This requires features like collaboration and version control, allowing several developers to work on projects. Many of the tools lack support for such features as they are not directly pointed out by the central idea of prompt engineering. Developers also require the flexibility to select different LLMs. We saw that most of the tools like PromptCharm or CoPrompt do not allow this, which inhibits the options of developers.

Our study also identified that tools also come in different structures and forms. Tools like PromptDeck, with its web-based user interface, provide developers with flexibility and ease of use while GenLine's macros-like structure offers targeted efficiency. These variations in tool structures can help developers tailor their workflows to specific needs, but also underscore the need for a unified platform that integrates essential features across use cases. Additionally, most of the tools like GenLine and GenForm serve a single use-case. While this can assist a certain population of developers, other developers also need tools that allow usage of diverse modalities. We have recently seen that tools like PromptLayer have started supporting multimodalities.

To address these gaps it is critical to revisit the findings of RQ1 and leverage the identified development life-cycle to support developers at every stage. Enhancing documentation through shared repositories and integrating robust collaboration tools could significantly improve the iterative refinement process. Additionally, RQ2 reveals that while tools like PromptLayer address many aspects of the life-cycle, they still lack features such as comprehensive problem framing. Incorporating these features along with artifacts like structured evaluation reports and reusable prompt templates would greatly enhance tool effectiveness.

4.1 Future Research

Advancing prompt engineering requires bridging gaps in workflows and tools by focusing on integrated environments that support the entire prompt engineering life cycle. Collaborative platforms with features like real-time co-editing, context-aware suggestions and automated version control could streamline workflows. Tools should also support multimodal capabilities and specialized use cases, such as healthcare or education, to enhance versatility and precision.

Improved documentation practices, including shared repos-

Tool Features \ Tools	GenLine and GenForm	CoPrompt	PromptCharm	Promptify	PromptDeck	PromptLayer
Problem Details	-	-	-	-	✓	-
Prompt Editor	✓	✓	✓	✓	✓	✓
LLM Selection	-	-	-	-	✓	✓
Iterative Refinement	✓	✓	✓	✓	✓	✓
Version Control	-	✓	✓	✓	-	✓
Collaboration	-	✓	-	-	-	✓
LLM Parameter Tuning	-	-	✓	✓	-	✓
Evaluation	-	✓	✓	✓	✓	✓
Custom Prompt Templating	-	-	-	-	-	✓
Considered Use Case	Code Generation		Text to Image Generation		Text Generation	

Figure 4: Evaluation of tools based on existing features

itories and structured reports, can boost reproducibility and collaboration. Ethical considerations must also be integrated to ensure responsible AI usage and prevent misuse. These advancements would make prompt engineering more accessible, reliable and effective across diverse applications.

5. CONCLUSIONS

Advancements in LLMs, alongside the use of natural language inputs, have enabled these models to address critical problems. Moving beyond simple experimentation, prompt developers now systematically engineer prompts, following a defined set of actions to enhance the prompt engineering process. This study explores the prompt development life cycle and identifies key activities such as problem framing, LLM selection and documentation, in addition to the iterative process of prompt design and refinement.

The study also highlights the various artifacts generated during prompt engineering and explains how they help optimize the process. Furthermore, we examine existing tools that have been developed over time to streamline prompt engineering, assessing how effectively these tools align with developer activities based on their offered features. Due to the novelty of this technology, these tools are still in their early stages.

However, trends indicate that they are updated continuously to incorporate features that keep pace with LLM advancements and evolving user interactions. For instance, PromptLayer shows promise for more complex projects. By analyzing prompt engineering activities, the study also identifies gaps in current tools. Finally, this paper serves as a guideline for prompt developers and as a reference for improving prompt engineering tools. By addressing the current state of prompt engineering and the limitations of available tools, this field can progress toward optimizing prompt engineering and enhancing tools that help developers fully leverage LLMs.

6. REFERENCES

[1] A. Ahmed, M. Hou, R. Xi, X. Zeng, and S. A. Shah. Prompt-eng: Healthcare prompt engineering: Revolutionizing healthcare applications with precision prompts. In *Companion Proceedings of the ACM Web Conference 2024*, WWW '24, page 1329–1337, New York, NY, USA, 2024. Association for Computing Machinery.

[2] J. M. Bass. Artefacts and agile method tailoring in large-scale offshore software development programmes. *Information and Software Technology*, 75:1–16, 2016.

[3] L. Beurer-Kellner, M. Fischer, and M. Vechev. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1946–1969, June 2023.

[4] S. Brade, B. Wang, M. Sousa, S. Oore, and T. Grossman. Promptify: Text-to-image generation through interactive prompt exploration with large language models. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, UIST '23, New York, NY, USA, 2023. Association for Computing Machinery.

[5] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners, 2020.

[6] A. Bucchiarone, M. Panciera, A. Cicchetti, N. Mana, C. Castelluccio, and L. Stott. Promptdeck: A no-code platform for modular prompt engineering. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, MODELS Companion '24, page 895–904, New York, NY, USA, 2024. Association for Computing Machinery.

[7] A. Carrera-Rivera, W. Ochoa, F. Larrinaga, and G. Lasa. How-to conduct a systematic literature review: A quick guide for computer science research.

MethodsX, 9:101895, 2022.

[8] P. Denny, V. Kumar, and N. Giacaman. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, page 1136–1142, New York, NY, USA, 2023. Association for Computing Machinery.

[9] P. Denny, J. Leinonen, J. Prather, A. Luxton-Reilly, T. Amarouche, B. A. Becker, and B. N. Reeves. Prompt problems: A new programming exercise for the generative ai era, 2023.

[10] I. D. Fagadau, L. Mariani, D. Micucci, and O. Riganelli. Analyzing prompt influence on automated method generation: An empirical study with copilot. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, ICPC '24, page 24–34. ACM, Apr. 2024.

[11] L. Feng, R. Yen, Y. You, M. Fan, J. Zhao, and Z. Lu. Coprompt: Supporting prompt sharing and referring in collaborative natural language programming. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, CHI '24, page 1–21. ACM, May 2024.

[12] A. J. Fiannaca, C. Kulkarni, C. J. Cai, and M. Terry. Programming without a programming language: Challenges and opportunities for designing developer tools for prompt programming. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI EA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[13] E. Jiang, E. Toh, A. Molina, A. Donsbach, C. J. Cai, and M. Terry. Genline and genform: Two tools for interacting with generative language models in a code editor. In *Adjunct Proceedings of the 34th Annual ACM Symposium on User Interface Software and Technology*, UIST '21 Adjunct, page 145–147, New York, NY, USA, 2021. Association for Computing Machinery.

[14] T. Kraljic and M. Lahav. From prompt engineering to collaborating: A human-centered approach to ai interfaces. *Interactions*, 31(3):30–35, May 2024.

[15] J. T. Liang, M. Lin, N. Rao, and B. A. Myers. Prompts are programs too! understanding how developers build software containing prompts, 2024.

[16] Y. Liu, G. Deng, Z. Xu, Y. Li, Y. Zheng, Y. Zhang, L. Zhao, T. Zhang, and K. Wang. A hitchhiker's guide to jailbreaking chatgpt via prompt engineering. In *Proceedings of the 4th International Workshop on Software Engineering and AI for Data Quality in Cyber-Physical Systems/Internet of Things*, SEA4DQ 2024, page 12–21, New York, NY, USA, 2024. Association for Computing Machinery.

[17] A. Mahdavi Goloujeh, A. Sullivan, and B. Magerko. Is it ai or is it me? understanding users' prompt journey with text-to-image generative ai tools. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, CHI '24, New York, NY, USA, 2024. Association for Computing Machinery.

[18] N. Nazar, Y. Hu, and H. Jiang. Summarizing software artifacts: A literature review. *Journal of Computer Science and Technology*, 31(5):883–909, 2016.

[19] S. Petridis, M. Terry, and C. J. Cai. Promptinfuser: Bringing user interface mock-ups to life with large language models. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI EA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[20] PromptLayer Team. Promptlayer: Prompt management and experimentation for llms, 2024. Accessed: 2024-12-14.

[21] L. Reynolds and K. McDonell. Prompt programming for large language models: Beyond the few-shot paradigm, 2021.

[22] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha. A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv preprint arXiv:2402.07927*, 2024.

[23] V. F. D. Santana. Challenges and opportunities for responsible prompting. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, CHI EA '24, New York, NY, USA, 2024. Association for Computing Machinery.

[24] S. Vatsal and H. Dubey. A survey of prompt engineering methods in large language models for different nlp tasks, 2024.

[25] A. Vijayan. A prompt engineering approach for structured data extraction from unstructured text using conversational llms. In *Proceedings of the 2023 6th International Conference on Algorithms, Computing and Artificial Intelligence*, ACAI '23, page 183–189, New York, NY, USA, 2024. Association for Computing Machinery.

[26] B. Wang, J. Liu, J. Karimnazarov, and N. Thompson. Task supportive and personalized human-large language model interaction: A user study. In *Proceedings of the 2024 Conference on Human Information Interaction and Retrieval*, CHIIR '24, page 370–375, New York, NY, USA, 2024. Association for Computing Machinery.

[27] J. Wang, Z. Liu, L. Zhao, Z. Wu, C. Ma, S. Yu, H. Dai, Q. Yang, Y. Liu, S. Zhang, et al. Review of large vision models and visual prompt engineering. *Meta-Radiology*, page 100047, 2023.

[28] Z. Wang, Y. Huang, D. Song, L. Ma, and T. Zhang. Promptcharm: Text-to-image generation through multi-modal prompting and refinement. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, CHI '24, New York, NY, USA, 2024. Association for Computing Machinery.

[29] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt, 2023.

[30] Q. Ye, M. Axmed, R. Pryzant, and F. Khani. Prompt engineering a prompt engineer, 2024.

[31] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba. Large language models are human-level prompt engineers, 2023.

The Role of Technical Debt in Legacy System Modernization: A Systematic Literature Review

Maren Hanke
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
maren.hanke@rwth-aachen.de

Tsvetina Angelova
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
tsvetina.angelova@rwth-aachen.de

ABSTRACT

Managing legacy systems comes with challenges such as high maintenance costs or a complex structure. The decision-making process for modernizing legacy systems is often not trivial, as financial and resource budgets are usually limited. Therefore priorities must be set on what to modernize and in what order. One problem that often arises with legacy systems is a large amount of technical debt. Technical Debt (TD) describes the trade-off made by adopting sub-optimal solutions for short-term benefits, potentially leading to higher costs and effort in the future. Modernization of legacy systems is typically desired and the knowledge of technical debt and the different types of TD that occur in a software system can be used to make decisions when modernizing. Currently, there is research on TD prioritization and legacy system modernization, however, there is no summary of TD prioritization approaches in legacy system modernization. Therefore, in this study, a systematic literature review was conducted, focusing on real-world case studies to systematically investigate the benefits of understanding TD in legacy system modernization. Key types of TD that impact modernization were identified, along with strategies for mitigating them. This review provides an overview of the current state of knowledge on the impact of TD on modernization and possible links between modernization and technical debt.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

Keywords

legacy system modernization, technical debt, systematic literature review, software quality, refactoring, software maintenance, code smells, software evolution, maintainability

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2024/25 RWTH Aachen University, Germany.

1. INTRODUCTION

In the rapidly evolving field of software development, Technical Debt (TD) has become a critical concept for understanding the relationship between short-term development benefits and system sustainability. The term TD refers to certain design or implementation choices that are made to make software development faster [30]. However, while they are beneficial in the short term, they often result in increased costs and challenges in the future. Some TDs are inevitable due to changing requirements or market pressures, and often impact the software lifecycle, affecting maintenance, scalability, and adaptability [15].

As the TD accumulates over time, it often leads to the emergence of legacy systems. This term refers to older software systems that, while essential to some business operations, become difficult to maintain, update, or integrate with newer technologies [21]. Legacy systems are common across industries due to their role in supporting critical operations. However, they present significant challenges, such as maintenance costs, reduced performance, and limited adaptability [13]. To address these issues, organizations often undertake modernization processes aimed at transforming legacy systems to meet current and future requirements.

Modernization involves improving the performance, adaptability, and cost-effectiveness of software systems. While modernization is a critical step in ensuring long-term software systems, it is not without its disadvantages. Organizations must navigate between achieving quality modernization and managing costs [22]. Understanding the role of the TD in this process is very important because it influences the reason for modernization and the way it is implemented.

While numerous studies have explored TD in software development [3, 6, 5, 9, 11], a comprehensive understanding of how it influences legacy system modernization is still lacking. In particular, the impact of different types of TD on the modernization process and how a better understanding of TD can support decision-makers in these projects has not yet been systematically examined. Few well-defined, empirically validated frameworks exist to guide practitioners in identifying, managing, and prioritizing TD in this context.

This gap between TD in broader software development and TD in legacy systems makes it difficult to define the specific causes and consequences of TD that obstruct modernization or to adopt effective strategies for addressing them. Without an understanding of this, decision-making in modernisation projects remains ad hoc and can increase the risk of inefficient processes.

This paper addresses this gap by conducting a Systematic

Literature Review (SLR), exploring the role of TD in legacy system modernization. To achieve this, the study identifies key types of TD that influence the modernization process and explores strategies to mitigate TD during modernization. By addressing these objectives, this paper provides a deeper understanding of TD's impact.

The remainder of the paper is structured as follows. First, in Section 2 a background on TD is given, followed by related work on TD and on legacy system modernization. Next, in Section 3 the methodology is described, detailing the research strategy. This is followed by Section 4 where the findings, which focus on the effects of TD on legacy system modernization, are discussed. The Section 5 examines possible implications that can be derived from the results. Finally, the paper concludes in Section 7 by summarizing the key findings and suggesting directions for future research.

2. BACKGROUND AND RELATED WORK

This section describes the core ideas of Technical Debt (TD), followed by an overview of research in the field of TD and legacy system modernization.

2.1 Background

In this Section, the term Technical Debt is explained to provide a foundation for the following Sections.

2.1.1 Technical Debt

The metaphor of TD was first introduced by Cunningham in 1992 [12]. Cunningham describes TD as a trade-off between high-quality, maintainable code that requires more effort initially but saves time in the future, and the short-term goal of deploying software quickly. Achieving faster delivery often means going into TD by sacrificing quality. This TD accrues interest every time the low-quality code is accessed since it requires more effort compared to working with a TD-free system. One can decide to keep the debt or to pay it off.

There are different views of what falls under the definition of TD and different approaches to classifying TD. Robert C. Martin [37] argues that a mess should not be considered because a TD has a benefit and a mess does not. Fowler [19] argues similarly, but does not distinguish between a debt and a mess, and instead focuses on categorizing TD into reckless and prudent debt. A reckless debt is one that is taken on lightly and without consideration and a prudent debt is one that was well-considered and carefully evaluated before being taken on. Fowler further distinguishes TD into deliberate and inadvertent debt, forming a technical debt quadrat. While deliberate debt is taken on knowingly, inadvertent debt is taken on unaware. This study considers all of Fowler's TD categories when examining the interplay between TD and modernization.

When discussing TD, one should also consider the related concept of Code Smells (CS). CSs refer to parts of the code that violate best practices and are easy to detect. However, unlike TD, CSs are not always problematic themselves but often rather indicate the presence of a problem. A very long method is an example of a CS that is not in itself a problem but can lead to problems [18].

2.1.2 Types of Technical Debt

In 2014, Li et al. [30] conducted a systematic mapping study to understand TD and summarize current research on its management. This study included publications between 1992 and 2013, analyzed 96 selected studies, and derived a classification of TD into ten categories and additional subcategories from these studies. Three of these ten identified categories are shown in Table 1. These three types will be reconsidered and their importance for this paper will be discussed in Section 5.

TD Type	Description
Architectural TD (ATD)	"Architecture decisions that make compromises in internal quality aspects".
Code TD	Code violating best practices.
Documentation TD	"Insufficient, incomplete, or outdated documentation in any aspect of software development."

Table 1: TD categories and descriptions as identified by Li et al. [30].

2.2 Related Work

In the following, a brief overview of related work in the field of TD and modernizing legacy systems is given, starting with Systematic Reviews (SRs) on TD and followed by SRs on the modernization of legacy systems.

2.3 Systematic Reviews of TD

Over the last decade, a significant amount of research on TD has been conducted. In this subsection, the focus lies on SRs on TD prioritization and management. Lenarduzzi et al. [28] published in 2021 an SLR that summarizes the TD prioritization approaches of 44 primary studies proposed until 2020. During their background research, they identified eleven SRs. Of these eleven SRs ten are presented in the following as one was not accessible. These are supplemented by more recent publications.

The earliest identified work is the case study by Tom et al. in 2012 [36], which proposed a theoretical framework for understanding TD. Over the last ten years, various studies have sought to identify TD identification approaches [5], categories of TD types [30, 33, 5], Technical Debt Management (TDM) strategies [30, 33, 5] and TD prioritization strategies [4, 27]. For instance, Becker et al. [8] conducted a SLR on TDM to identify how trade-off decisions in TDM are made in current literature. Meanwhile, Ampatzoglou et al. [7] explored financial approaches to TDM in a SLR. Fernández-Sánchez et al. [16] focused in their systematic mapping study on identifying elements critical for TDM. They grouped these elements into three categories, mapped them to three stakeholders' perspectives, and revealed that TDM is highly context-dependent.

Other studies focused on TD in specific contexts. For example, Behutiye et al. identified in a SLR 12 TDM strategies in the context of agile software development, while the systematic mapping study by Bogner et al. identified TD types, antipatterns, and solutions for TD in the context of AI-based systems. Toledo et al. conducted a multiple case study to identify Architectural TD (ATD), its costs, impacts, common solutions, and relationships in the context of microservices. Besker et al. [10] developed a descriptive model to characterize and explain ATD, aiming to spread stakeholder awareness of ATD.

2.4 Systematic Reviews in Legacy System Modernization

This subsection reviews SRs and exploratory studies on legacy system modernization, focusing on widely cited publications.

In 2014, Khadka et al. [25] conducted an exploratory study by interviewing 26 industry practitioners to uncover the key drivers and challenges of system modernization. Agilar et al. [2] followed with a mapping study in 2016, to identify processes, techniques, and tools for system modernization. The SLR by Abdellatif et al. [1] developed a taxonomy and classification of existing service identification approaches for modernizing legacy systems to a service-oriented architecture. Furthermore, Wolfart et al. [40] have dedicated their study to creating a roadmap for modernizing monolithic legacy systems into microservices. Another roadmap was developed by Khadka et al. [25] who advocate a modernization framework to take advantage of cloud computing.

2.5 Research Trends and Gaps

In their research, Holvitie et al. [23] found 31 publications addressing the concepts of TD and legacy. Of these, five publications explicitly highlighted the difference between TD and legacy, while the remaining 26 publications compared the two concepts, discussed legacy as a contributing factor to TD, or indicated that legacy underscores the negative effects of TD. This emphasizes the importance of further exploring the interplay between these concepts and examining the potential use of the insights on their relation for modernization purposes. While many SRs have explored TD and legacy system modernization, there is a lack of exploration into their relation or the identification of suitable TDM strategies for modernization projects. This SLR aims to address this gap.

3. METHOD

To conduct this study, the guidelines for Systematic Literature Review (SLR) developed by Kitchenham [26] were followed. A SLR provides a structured approach to identifying, evaluating, and interpreting all available research relevant to a specific research question or topic area of interest [26]. In addition, the Snowballing process, as defined by Wohlin [39], was applied.

The research process is described in the following subsections. First, the research questions are provided, followed by the search strategy, the inclusion and exclusion criteria, and a quality assessment.

3.1 Definition of Research Questions

The purpose of this study is to identify, categorize, and examine the role of TD in legacy system modernization, according to recent literature. The study was guided by the following research questions:

- **RQ1:** What types of TD accelerate a system's transition to a legacy system?
- **RQ2:** What strategies are employed in existing projects for considering TD during modernization?
- **RQ3:** What challenges exist in managing TD, and how can they guide best practices for legacy system modernization?

The first research question (**RQ1**) focuses on the types of TD that accelerate the process of a system evolving into a legacy system. The second aims to explore how TD can be prioritized and what strategies exist to handle TD in legacy system modernization projects successfully (**RQ2**), while the last research question focuses on the challenges and their possible solutions in TD management and what insights they provide for modernization (**RQ3**).

These questions are addressed to provide insights into the interplay between TD and legacy system modernization and to identify TD management strategies that have been empirically validated to be effective in real-world modernization projects.

3.2 Search Process

The search process included identifying the most relevant bibliographic sources and search terms, defining the criteria for inclusion and exclusion, and outlining the selection process for making inclusion decisions.

The databases in Table 2 were used for the search process, as they are the most relevant for the field of computer science.

Database	Date of Search	# Papers
dblp	01.11.2024	58
IEEExplore	31.10.2024	53
IET Digital Library	02.11.2024	38
Scopus	24.10.2024	183
SpringerLink	03.11.2024	26

Table 2: Databases used for the search process.

Due to the initial high number of search results in SpringerLink, which were likely to contain many irrelevant entries, the search in SpringerLink was restricted to the discipline of computer science, using only the search term "Technical Debt Legacy System Modernization" and limiting the results to conference papers and articles, resulting in more relevant and manageable results. Furthermore, for all databases, the option to display results containing all search term words was applied. For the databases, the following search terms were used:

- Technical Debt and Legacy System Modernization
- Technical Debt and Software Modernization
- Technical Debt and Legacy Systems
- Technical Debt Prioritization

Various search terms were initially tested in the databases, and only those yielding results relevant to the topic were retained.

3.3 Inclusion and Exclusion Criteria

The inclusion and exclusion criteria ensure the selection of relevant and high-quality studies for this research. Studies that discussed strategies to handle TD, with a focus on TD management and the effects on maintainability, were included, as well as studies providing approaches for removing or refactoring TD.

Studies were excluded if they did not fit the research topic, did not provide answers to the research questions, were not written in English, or did not contain peer reviews, such

as blog posts or forum discussions. Duplicate studies were identified and only the most recent and comprehensive version was included in the database. In addition, publications where access to the full text was not possible were not included. In cases where journal articles extended conference papers based on the same dataset, only the most recent version was included in the analysis.

3.4 Quality Assessment

After an initial review of the 358 works identified in the search, each one was checked against the defined inclusion and exclusion criteria. After this process, 14 studies remained that met the requirements and provided insights into the role of TD in legacy system modernization. The results of this process are reported in Table 3, which includes the number of papers excluded due to various criteria, along with the number of papers that remained.

Step	# Papers
Retrieved from databases	**358**
False positive	81
Discarded by language	1
Discarded by title	38
Discarded by abstract	28
No access or no pdf available	33
Duplicates	72
Remaining	**105**
Discarded by full-text	91
Remaining studies	**14**

Table 3: Results of search and application of quality assessment criteria.

4. RESULTS

This section examines the findings from the analysis of the 14 primary studies. First, the potential of addressing TD to mitigate some of the challenges of legacy system modernization is explored. This is followed by a discussion of the strategies for prioritizing TD. Finally, the challenges in TD management and the sources of TD accumulation are evaluated to understand how to prevent TD accumulation during modernization.

Holvitie et al. [23, 24] found in their survey of 69 TD instances, with 184 responses, that the origin of components carrying TD was 76% perceived to be in software legacy and only 15% not to be rooted in legacy, suggesting a strong connection between TD and legacy software.

4.1 Challenges of Legacy System Modernization

Khadka et al. [25] identified the challenges of legacy system modernization. Some of these challenges are presented below, along with a discussion of how reducing TD can potentially mitigate them.

A complex architecture was identified as a challenge, which also complicates testing, which is seen as another challenge of modernization. This problem could potentially be mitigated by reducing ATD prior to modernization. Inadequate documentation of the legacy system is also mentioned as a problem, as this often leads to a lack of understanding of the functionality of the components and makes it difficult to extract the business logic. It is noted that it can be beneficial to document the legacy system before starting the modernization, which is equivalent to reducing the documentation debt. Another challenge is to communicate the reasons and consequences of legacy system modernization. Again, the concept of TD could be helpful as it makes it easier to communicate the consequences and benefits of retaining or repaying TD. These challenges suggest that refactoring to reduce certain types of TD prior to modernization could be beneficial in mitigating some challenges of modernization. Several studies report the successful reduction of TD during refactoring, which are reviewed below.

4.2 Strategies for Addressing TD During Modernization

During the SLR, no studies were found that directly investigated the role of TD in legacy systems modernization. Therefore, the focus is on studies that present strategies to reduce TD during refactoring which can be applied prior to modernization to mitigate the challenges of legacy system modernization. Another area of focus is studies that examine the role of TD during refactoring or the transition to a different architectural style.
First, studies focusing on TD in general are listed, followed by studies that examine specific types of TD. Then, studies on the influence of TD during architectural style transitions are presented.

4.2.1 Overview of General TD Studies

During modernization, it can be useful to exchange one debt for another with a lower interest rate. Buschmann [11] gave an example where a company opted for poor software modularization to speed up development. Unexpected performance problems led to the interpreter being replaced by a run-time compiler, which solved the performance problems but reduced runtime flexibility. This is one example of a debt being traded for another with a lower interest rate. It illustrates that considering the concept of TD during modernization can help to choose a strategy that minimizes future interest while focusing on the gains.

The study by Gupta et al. [21] provides an example of how TD can be successfully reduced in a legacy system. They continuously monitored and prioritized TD to pay back incurred and new TD. The approach was split into four steps: First, the TD items are identified, then a strategy is developed to repay the TD and prevent new TD from arising. This is followed by the execution, where the importance and urgency of the TD items are determined, and in the second step, the impact of each TD item on business value and the effort required to address it are assessed. This was followed by a validation phase. Although the focus was not on modernization but on managing TD in a legacy system, it demonstrates the need and benefits of considering TD when dealing with legacy systems.

Chowdary et al. [20] recalled another successful reduction of TD. They present a method for refactoring that significantly reduces the Self-Admitted Technical Debt (SATD), which refers to TD explicitly reported by developers. They introduce a framework that uses NLP word embeddings to identify and prioritize SATD. The proposed method showed higher accuracy, precision, detectability, and F1 score than existing models, and after refactoring, SATD instances were significantly reduced.

4.2.2 Specific Types of TD and Their Impact

Several studies focused on specific types of TD. In the primary studies reviewed, Architectural Technical Debt (ATD) (21%) and Code Debt (7%) are the most commonly reported types of TD. The following studies focused on recommending guidelines for refactoring. Their findings have the potential to provide valuable insights if refactoring is performed to reduce TD prior to modernization.

An example of effective TD management in refactoring is the Code Smell Intensity Index by Fontana et al. [17]. The Code Smell Intensity Index prioritizes which CSs are most critical and should be removed first. CSs are not a problem in themselves, but they indicate problems and can be a source of TD. The experimental evaluation showed that only 10% of the CSs are assigned to the highest of the five defined intensity levels, which significantly reduces the number of instances to be inspected and thus saves time during refactoring. However, the index does not consider whether the smells are related. If smells are related or co-related, they become more critical since they can lead to architectural smells, therefore they require earlier analysis.

As we have seen, a key challenge in modernizing legacy systems is a complex system architecture [25]. Reducing ATD can potentially mitigate this challenge by improving the system architecture. Sas et al. [35] examined which Architectural Smells (AS) should be prioritized, providing suggestions that can help with refactoring. They explored AS in 524 versions across 14 different projects. As with CSs, ASs are not a problem themselves but can indicate a problem and thus a TD. They considered three different types of AS in their study: Unstable Dependency (UD), Hub-like Dependency (HL), and Cyclic Dependency (CD). Sas et al. concluded that small debt items should not be prioritized during refactoring, as they are likely to be intentional. Instead, old complex TD items should be prioritized, as they can have a negative impact on the important parts of the system and their maintainability. Recently introduced CD smells should also not be prioritized, as they will usually disappear in future releases. CD smells are cyclic dependencies formed by many components. Instead, the focus should be on HL and UD smells as they tend to be more persistent. UD smells are components that depend on many less stable components. The more unstable components the main component depends on, the more likely it is to change, potentially causing a ripple effect, as components dependent on the main component will probably also need to be changed. HD smells are components with more ingoing and outgoing dependencies than the median in the system. This structure complicates maintenance and increases change effort, as highly coupled central components are overburdened with responsibility. HL smells are a good choice for refactoring because addressing them helps to reduce complexity and future maintenance efforts. Since refactoring of hub-like dependencies typically focuses on one main component, it is easier than addressing CD smells that affect multiple components.

Martini et al. [31] discovered ATD elements that cause other system parts to be contaminated with the same problem, which can lead to non-linear growth of interest. One type of debt identified is contagious debt, which can be caused by dependency violations and unrecognized dependencies as the system grows. For instance, a database without a standardized interface can spread debt to other components that access the database. This rapidly increases costs as the components grow, as refactoring requires changes to all affected components. Regular monitoring and architectural retrospectives are essential to identify ATD items early and prevent them from spreading. According to Martini et al., the most dangerous ATD items are hidden ATD and incomplete refactorings. An example of incomplete refactoring is inconsistent communication patterns that should be replaced by a new pattern, but due to hidden side effects and time pressure, the old patterns are left alongside the latest pattern, leading to more ADT. Weak awareness of debt and time pressure during refactoring can create cycles of events that lead to accumulating TD.

4.2.3 TD in Architectural Style Transitions

The following studies focused on the effect of the transition to a different architectural style on the TD.

The following two studies [13, 29] reported a long-term decrease in TD: Justino et al. [13] describe a process for migrating legacy systems to a service-oriented architecture that minimizes risk by applying the SPReaD process while ensuring system quality improvement throughout the migration process. As a result, the TD was reduced by 47%.

Lenarduzzi et al. [29] observed a similar long-term reduction in TD in a four-year case study of a project transitioning from a monolithic architecture to microservices, the TD was monitored in parallel. During the development of microservices, the TD initially increased. Afterward, the TD grew slower than in the original monolithic system, resulting in a long-term decrease in TD compared to the monolithic system.

While transitioning to a different architectural style can result in a long-term TD decrease [13, 29], the study by Toledo et al. [14] emphasized that there is also a risk of accumulating microservice-specific ATDs during the migration. Their case study of three large companies in the early stages of migration to microservices explored the influence on ATD. ATD can be costly if the debts are not or are too late identified, avoided, or removed. During the early stages of migration, microservice-specific ATDs (MS-ATDs) may accumulate. For example, practitioners may continue using poorly defined APIs in microservices while attempting to maintain compatibility with old functionalities. One of the reasons for companies to migrate to microservices is to repay known ATDs from their previous architectures while, at the same time, obtaining the benefits of this new architectural style. This brings the risk of trading old known ATDs for new unknown MS-ATDs. This reduces the benefits and may prove to be more costly than previous debts. Toledo et al. [14] identified misusing shared libraries, sharing databases during migration, and microservice coupling as the most important MS-ATD items to look out for and suggest that MS-ATD needs to be identified and handled early on to avoid accumulating costs.

One commonly mentioned reason for transitioning to a new architecture is eliminating the TD of the old system. However, while a long-term decrease of TD can be observed in two studies [13, 29], Toledo et al. [14] highlighted the risk of taking on new unknown TD during the process. This underlines the importance of the awareness of TD during the architecture transition and understanding the TD items specific to the target architecture style.

4.3 Existing Challenges in Managing TD

In the following, an overview of existing challenges in TD management and sources of TD accumulation is presented. Awareness of the sources during modernization holds the potential to mitigate TD accumulation during modernization.

Vogel-Heuser and Rösch [38] discuss the challenges of time pressure in software installed in automated production systems, where tight deadlines and penalties can lead to compromised architectural principles. Time is a recurring challenge in dealing with ATD, as highlighted by Mo et al. [32], who emphasize its importance in the management process. Weak awareness of debt and time pressure during refactoring can create circles of events that lead to accumulating TD [31]. Several papers indicated that modernization comes at the risk of increased TD if not done properly [14, 31]. TD has to be identified early, hence an awareness during the modernization of potentially introducing new TD is needed. Regular monitoring is necessary for early detection and prevention of the spreading of TD [31]. Early detection and refactoring of ATD, as Xiao et al. [41] suggest, could significantly reduce maintenance costs. Furthermore, during modernization, it is essential to complete each refactoring, since incomplete refactorings are a cause for the accumulation of new TD [31].

Researchers focused on different perspectives on the causes of TD. Ernst et al. [15] surveyed 1831 participants who revealed architectural decisions as the most important source of TD. Architectural issues are often caused by decisions made early in the software lifecycle and may remain unnoticed until the late stages. This aligns with the findings of Martini et al. [31] according to which one of the most dangerous ATD items is hidden ATD. Leading to the assumption that during a modernization it can be beneficial to pay attention to architectural decisions and the hidden ATDs they may introduce.

In the interview-based case study by Rios et al. [34] with 10 participants, the causes for the accumulation of TD were analyzed. The most frequently cited causes were deadlines which aligns with the concerns several researchers had regarding time pressure as discussed above [38, 32, 31], lack of knowledge, inadequate planning, and lack of maturity to follow the process. This indicates that during modernization awareness of these causes can help in preventing TD to arise during modernization.

Resuming, the main causes for the accumulation of ATD are time pressure, lack of ATD awareness, lack of knowledge, insufficient planning, and incomplete refactoring. In addition, architectural decisions should be made carefully to avoid the introduction of hidden ATD. Paying attention to these causes during modernization could prevent TD accumulation.

5. DISCUSSION

This section discusses the results and potential implications of which role TD should play in legacy system modernization. Although this field appears to be relatively young compared to established areas like software testing or software quality, it seems like it has seen significant contributions over the past decade, with growing activity and interest among researchers.

While there is no direct proof of the influence of TD on modernizing legacy systems, we believe that taking TD into account when modernizing legacy systems is beneficial because it helps gain additional insights as our findings indicate. A strong connection between TD and legacy software is suggestive since the origin of components carrying TD is 76% perceived to be in software legacy [23, 24].

Because of the challenges legacy system modernization holds we assume that refactoring a system to reduce ATD and Documentation TD before modernization could help in reducing these challenges. We have seen several examples of refactoring successfully reducing TD underlining that there are TD management strategies that can be used for TD reduction prior to modernization. It was noticeable not all TDs types introduced by Lie et al. [30] (see Section 2) were addressed in the reviewed studies. We observed that only Architectural, and Code Debt appeared, indicating that these are the types of TDs that have been perceived by other research as most important. The modernization challenges summarized by Khadka et al. [25] let us imply that Code TD has minimal impact in addressing modernization challenges. This indicates that managing Code Debt is only beneficial in refactoring but not in modernization. We assume that the reason could be that Code Debts are paid back during modernization but their appearance does not cause challenges like ATDs do. Repaying Documentation TD which as repaying ATD has the potential to reduce modernization challenges is not mentioned in the examined studies. We suspect that the reason could be that removing Documentation TD is more straightforward and hence did not raise the researcher's interest.

An example showed that legacy systems can benefit from TD reduction [21] even when not being modernized, indicating that TD management should play a role when improving legacy systems' performance and maintenance.

Contagious debt and AS and ATD items with dependencies on other components that potentially cause ripple effects were perceived as problematic. These TD items can result in high maintenance efforts and contagious debt can lead to nonlinear growth of interest [31]. Based on this, we presume that the accumulation of contagious debt and AS and ATD items with cyclic dependencies should be closely monitored and avoided during modernization. Furthermore, already existing contagious debt should be managed to help prevent TD from spreading and the system from degrading back to a legacy system.

We found that the main causes of TD accumulation are time pressure, lack of ATD awareness, lack of knowledge, insufficient planning, and incomplete refactoring. We suspect that awareness of these causes during modernization could help to prevent TD accumulation and hence the need for repeated modernization cycles.

Some legacy systems are migrated to other architectures. Two examples we have seen are the mitigation to microservices [29, 14] and to a service-oriented architecture [13]. These modernization efforts can initially increase TD. However strategic approaches such as early identification of TD and applying structured migration processes like SPReaD can reduce the TD in the long run [29, 14, 13]. But while transitioning to a new architecture is eliminating the TD of the old system it comes at the risk of introducing new unknown TD items [14]. Hence we suspect that awareness of TD during the architecture transition and understanding the TD items specific to the target architecture style is essential to ensure a long-term decrease of TD.

6. THREATS TO VALIDITY

This SLR may be subject to certain threads to validity. One limitation is the relatively short research period of three months, which limited the depth of the review process. To manage the large number of search results, filters were applied and the search was restricted to a set of keywords, potentially excluding relevant studies. Furthermore, to ensure the accuracy of the results, peer-reviewed publications were prioritized, including journals and conference proceedings. Despite these efforts, the quality of the results of this SLR relies on the quality and completeness of the included sources. This SLR focused on more recent papers to ensure the reflection of the current research in the field.

7. CONCLUSION

This section summarizes the findings and provides recommendations for future research.

From a software lifecycle perspective, it is important to understand and proactively manage TD. This work aims to provide in-depth research on the role of TD in legacy system modernization. The analysis of significant contributions in the field shows stable interest and activity among researchers despite the relative youth of this subject compared to more established fields like software development, testing, or quality.

This study investigated the role of TD in the modernization of legacy systems. It was found that there is likely to be a strong link between these areas and that eliminating ATD and Documentation TD prior to modernization can potentially reduce the overall challenges of modernizing legacy systems. The study highlighted the main causes of TD accumulation to avoid during modernization to help prevent repeated modernization cycles, as well as the TD elements that are considered particularly problematic, and to which developers, therefore, should pay particular attention during modernization. The study also showed that migrating to other architectures can reduce TD in the long term. However, awareness of TD during the architectural transition and understanding the TD elements that are specific to the style of the target architecture are essential. An example showed that legacy systems can benefit from TD reduction not only during modernization.

This research has investigated the findings from the interplay between TD and legacy system modernization, but it also highlights several areas requiring further investigation. There is still a lack of studies examining the benefits of considering TD during modernization. The observations of this work and the implications for modernization derived from them need to be tested in real-world case studies. Their benefits need to be evaluated against time constraints and the budget of modernization projects.

This study highlights the potential an awareness of TD during modernization holds.

8. REFERENCES

[1] M. Abdellatif, A. Shatnawi, H. Mili, N. Moha, G. E. Boussaidi, G. Hecht, J. Privat, and Y.-G. Guéhéneuc. A taxonomy of service identification approaches for legacy software systems modernization. *Journal of Systems and Software*, 173:110868, 2021.

[2] E. Agilar, R. Almeida, and E. Canedo. A Systematic Mapping Study on Legacy System Modernization. In *Proceedings of the 28th International Conference on Software Engineering and Knowledge Engineering*, International Conferences on Software Engineering and Knowledge Engineering, pages 345–350. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2016.

[3] M. Albarak and R. Bahsoon. Prioritizing technical debt in database normalization using portfolio theory and data quality metrics. In *Proceedings of the 2018 International Conference on Technical Debt*, pages 31–40, 2018.

[4] R. Alfayez, W. Alwehaibi, R. Winn, E. Venson, and B. Boehm. A systematic literature review of technical debt prioritization. In *Proceedings of the 3rd international conference on technical debt*, pages 1–10, 2020.

[5] 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.

[6] 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.

[7] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou. The financial aspect of managing technical debt: A systematic literature review. *Information and Software Technology*, 64:52–73, 2015.

[8] C. Becker, R. Chitchyan, S. Betz, and C. McCord. Trade-off decisions across time in technical debt management. In R. L. Nord, F. Buschmann, and P. Kruchten, editors, *Proceedings of the 2018 International Conference on Technical Debt*, pages 85–94, New York, NY, USA, 2018. ACM.

[9] W. N. Behutiye, P. Rodríguez, M. Oivo, and A. Tosun. Analyzing the concept of technical debt in the context of agile software development: A systematic literature review. *Information and Software Technology*, 82:139–158, 2017.

[10] T. Besker, A. Martini, and J. Bosch. Managing architectural technical debt: A unified model and systematic literature review. *Journal of Systems and Software*, 135:1–16, 2018.

[11] F. Buschmann. To pay or not to pay technical debt. *IEEE software*, 28(6):29–31, 2011.

[12] W. Cunningham. The wycash portfolio management system. *SIGPLAN OOPS MESS.*, 4(2):29–30, 1993.

[13] Y. de Lima Justino and C. E. da Silva. Poster: Reengineering legacy systems for supporting soa: A case study on the brazilian's secretary of state for taxation. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 125–126, 2018.

[14] S. S. De Toledo, A. Martini, P. H. Nguyen, and D. I. Sjøberg. Accumulation and prioritization of architectural debt in three companies migrating to microservices. *IEEE Access*, 10:37422–37445, 2022.

[15] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton. Measure it? manage it? ignore it? software practitioners and technical debt. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software*

Engineering, ESEC/FSE 2015, page 50–60, New York, NY, USA, 2015. Association for Computing Machinery.

[16] C. Fernández-Sánchez, J. Garbajosa, A. Yagüe, and J. Perez. Identification and analysis of the elements required to manage technical debt by means of a systematic mapping study. *Journal of Systems and Software*, 124:22–38, 2017.

[17] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda. Towards a prioritization of code debt: A code smell intensity index. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pages 16–24. IEEE, 2015.

[18] M. Fowler. Code smell. `https://www.martinfowler.com/bliki/CodeSmell.html`, 02/12/2024.

[19] M. Fowler. Technical debt quadrant. `https://martinfowler.com/bliki/TechnicalDebtQuadrant.html`, 25/09/2024.

[20] S. M. C. G and P. K. R. Automated identification and prioritization of self-admitted technical debt using nlp word embeddings. In *2023 International Conference on Self Sustainable Artificial Intelligence Systems (ICSSAS)*, pages 963–971, 2023.

[21] R. K. Gupta, P. Manikreddy, S. Naik, and K. Arya. Pragmatic approach for managing technical debt in legacy software project. In *Proceedings of the 9th India Software Engineering Conference*, pages 170–176, 2016.

[22] G. Hogan, P. Shalkauskaite, M. Zhu, M. Derwin, M. Yilmaz, A. McCarren, and P. M. Clarke. Investigating systems modernisation: Approaches, challenges and risks. In *European Conference on Software Process Improvement*, pages 147–162. Springer, 2024.

[23] J. Holvitie, S. Licorish, A. Martini, and V. Leppänen. *Co-Existence of the'Technical Debt'and'Software Legacy'Concepts*. 2016.

[24] J. Holvitie, S. A. Licorish, R. O. Spínola, S. Hyrynsalmi, S. G. MacDonell, T. S. Mendes, J. Buchan, and V. Leppänen. Technical debt and agile software development practices and processes: An industry practitioner survey. *Information and Software Technology*, 96:141–160, 2018.

[25] R. Khadka, B. V. Batlajery, A. M. Saeidi, S. Jansen, and J. Hage. How do professionals perceive legacy systems and software modernization? In P. Jalote, L. Briand, and A. van der Hoek, editors, *Proceedings of the 36th International Conference on Software Engineering*, pages 36–47, New York, NY, USA, 2014. ACM.

[26] B. Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele Univ.*, 33, 08 2004.

[27] V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, and F. A. Fontana. Technical Debt Prioritization: State of the Art. A Systematic Literature Review.

[28] V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, and F. A. Fontana. A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools. *Journal of Systems and Software*, 171:110827, 2021.

[29] V. Lenarduzzi, F. Lomio, N. Saarimäki, and D. Taibi. Does migrating a monolithic system to microservices decrease the technical debt? *Journal of Systems and Software*, 169:110710, 2020.

[30] Z. Li, P. Avgeriou, and P. Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

[31] 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. IEEE, 2015.

[32] R. Mo, J. Garcia, Y. Cai, and N. Medvidovic. Mapping architectural decay instances to dependency models. In *2013 4th International Workshop on Managing Technical Debt (MTD)*, pages 39–46, 2013.

[33] N. Rios, M. G. de Mendonça Neto, and R. O. Spínola. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. *Information and Software Technology*, 102:117–145, 2018.

[34] N. Rios, R. Oliveira Spinola, M. G. de Mendonca Neto, and C. Seaman. A Study of Factors that Lead Development Teams to Incur Technical Debt in Software Projects. pages 429–436. IEEE, 2018.

[35] D. Sas, P. Avgeriou, and F. A. Fontana. Investigating instability architectural smells evolution: an exploratory case study. In *2019 IEEE International Conference on software maintenance and evolution (ICSME)*, pages 557–567. IEEE, 2019.

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

[37] Uncle Bob. Clean Coder - A Mess is not a Technical Debt. `https://sites.google.com/site/unclebobconsultingllc/a-mess-is-not-a-technical-debt`, 10/12/2024.

[38] B. Vogel-Heuser and S. Rösch. Applicability of technical debt as a concept to understand obstacles for evolution of automated production systems. In *2015 IEEE International Conference on Systems, Man, and Cybernetics*, pages 127–132, 2015.

[39] 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*, pages 1–10, 2014.

[40] D. Wolfart, W. K. G. Assunção, I. F. Da Silva, D. C. P. Domingos, E. Schmeing, G. L. D. Villaca, and D. d. N. Paza. Modernizing Legacy Systems with Microservices: A Roadmap. In R. Chitchyan, J. Li, B. Weber, and T. Yue, editors, *Evaluation and Assessment in Software Engineering*, pages 149–159, New York, NY, USA, 2021. ACM.

[41] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng. Identifying and quantifying architectural debt. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 488–498, New York, NY, USA, 2016. Association for Computing Machinery.

Evaluating Automated Testing Methods for Microservice Systems

Jonas Hartwig
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
jonas.hartwig@rwth-aachen.de

Ahmad Al Housseini
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
ahmad.al-housseini@rwth-aachen.de

ABSTRACT

Microservices (MS) are an architectural style in which applications are composed of small, independent services that communicate over lightweight protocols, offering flexibility and scalability, but also introducing challenges in testing due to their distributed nature. To reduce the effort needed to deploy changes within an microservice architecture (MSA) system, automated testing paradigms may be used. They reduce the overall time developers have to spend on the testing process as a whole.

In this paper, we compare the different automated testing approaches used for MS and evaluate these approaches on the basis of their ability to properly test MSA systems. To achieve this comparison, we first provide an overview of microservice architectures and their testing strategies. We then examine the automation processes and assess these approaches in terms of robustness, efficiency, test coverage and reusability. By applying this methodology, we generate a ranking of some relevant testing methods. Unlike previous studies, this research focuses on identifying strengths and weaknesses of each approach, highlighting that no single testing methodology is universally superior. Instead, the most effective strategy combines multiple testing paradigms.

Keywords

Microservice Systems, Automated Testing, MSA

1. INTRODUCTION

Microservice-based architecture (MSA) is driven by architectural principles and involves the construction of complex systems from small loosely coupled components that communicate with each other through APIs. Each component is an independent deployable service. Multiple individual components are organised around specific business domains, providing flexibility, scalability and technological heterogeneity. Each service encapsulates its unique functionality and data while communicating with other services through APIs or event-driven integration[14]. An example of microservices architecture is presented in Figure 1. Independent services interact with each other or even external services to realize several functionalities within a distributed system.

Automated testing is an integral part of modern software development, which enables the efficient, scalable and reproducible evaluation of the software capabilities throughout the complete development life cycle. Using specialized tools, it runs and controls test scripts and cases which raise the level of simplicity in some testing areas. Automated testing saves manual effort and reduces the risk of errors due to forgetting or wrongly executing test cases. This increases reliability and stability. It is an important aspect because the complexity of software systems is increasing steadily. [6]

However, the automatization of the test process is a complicated matter. Microservice ecosystems and architectures are very complex, with a production system comprising hundreds of services. For instance, Netflix uses over 1,000 of them. Moreover, there currently doesn't exist an approach that provides a satisfactorily reliable test result. Another big issue is the lack of proper tools. (see Section 3 for more info) [21]

This paper consists of multiple sections. First we present our methodology in Section 2. In Section 3, we discuss the challenges facing automated testing of microservices. In Section 4 we provide an overview over some existing approaches. In Section 5, we compare and evaluate these approaches. Section 6 presents the threats to the validity of this work and finally we conclude and summarise in Section 7.

2. METHODOLOGY

The main goal of this work is to assess the capabilities of different automated testing approaches for microservice systems. We therefore pull knowledge about these different approaches from multiple papers. We then seek to assess the viability each approaches given several metrics. The assessment is purely based on the results of the application of each method that we are able to find and our opinion on how each approach could grant advantages or disadvantages regarding each metric. The metrics we use are taken from an IEEE glossary [1] and semantically adjusted to the context of this work.

To find papers regarding different testing approaches, we first seek to gain an overview over microservice testing. Papers about this topic will be searched for generally. They can then be used as indicators for closer examination of singular methods. Table 1 contains a list of all papers that we

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2018/19 RWTH Aachen University, Germany.

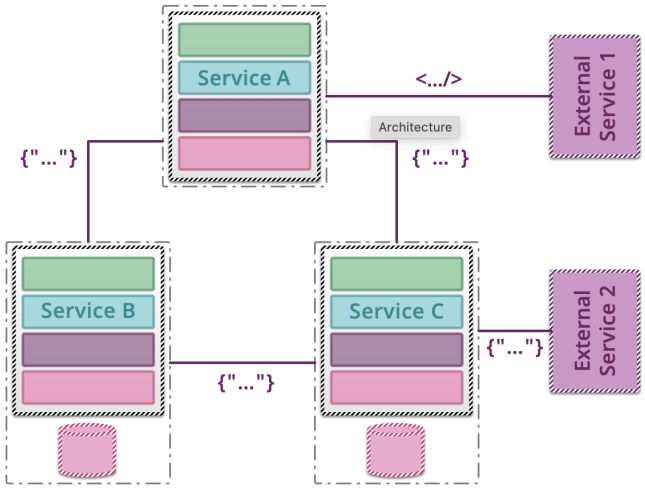


Figure 1: Multiple services work together as a system[4]

found by using queries on different search engines. These papers are primarily used for the overview over the topic. The papers used for the overview often already contain references to other papers useful to us. Those are not included in this table but listed in the end with all sources used.

We used IEEExplore and Google Scholar as search engines and queried either with a focus on microservices or automated testing. We used the following queries represented as regular expressions:

- microservice (architecture)? (pattern | testing | test automation)
- automated testing (of microservice architecture | robustness)

Not all of the possible queries yielded new useful results in either search engine used. Table 1 lists only the papers we were able to identify as useful for this work.

Some of the papers used as overview, list a high number of individual MSAs and testing approaches, often in combination with a number of uses in a certain context. We limited our analysis only to approaches that occurred more often to stay within the scope of this work.

After evaluating each testing method for our criteria we assign ranks based on our perception of their viability. We use these rank each criterion to calculate an overall score and reason about possible future direction and effectiveness of each method.

3. CHALLENGES IN AUTOMATED TESTING OF MICROSERVICES SYSTEMS

Waseem et al. [21] summarized the most prominent challenges found in MSA applications by using several other papers and their results. They identified as main challenge for automated testing a complex deployment, meaning that an increasing number of services are used together, especially if they are of different type. They further identify seven challenges which apply to MSA-based applications generally and therefore also apply to automated test generation.

Inter-communication testing concerns correct data transmissions between a potentially large number of services which in turn hinders proper integration and system-level testing. *Test feedback* describes procedures with performance evaluations early in development. Existing testing frameworks provide insufficient mechanism to reliably provide such feedback based an communication, file access and the underlying database. The primary issue with *integration testing* is generating effective integration test cases and analysing the generated logs. *Acceptance testing* is troubled by tools that support it for MSA applications in behaviour-driven development. *Granularity testing* is subject to asynchronicity complexity and managing chaining interfaces. *Performance testing* as an evaluation attempt for system behaviour and performance, not functionality, suffers from unclear or changing metrics used. *Runtime testing* suffers from the high flexibility inherent to MSA and the evolving nature of an MSA system.

Abdelfattah et al. [7] conducted a study regarding test coverage metrics for E2E (End-to-End) testing. They describe a problem regarding the discrepancy of perceived coverage by a tester and the actual coverage of an MSA system. Verification mechanisms for testing the complete state of a system are often missing.

4. OVERVIEW OVER EXISTING TEST AUTOMATION APPROACHES

In this section we provide a small overview over some of the existing test approaches. Waseem et al. [21] provided valuable literature and references, which were helpful in defining the testing approaches.

Figure 2 shows the hierarchical structure of the different test categories and how the system needs to maintain a strong base of comprehensive tests, like unit and component tests, for efficiency and reliability purposes while reducing the number of integration and end-to-end tests for optimized execution time and system coverage. The extents of a selection of test approaches have been visualised in Figure 3.

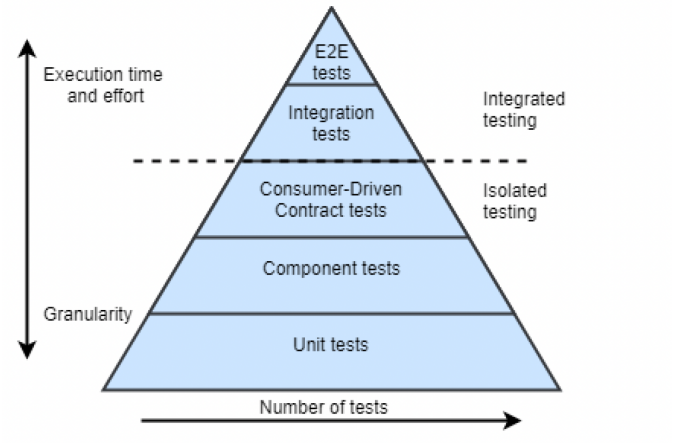


Figure 2: Test pyramid with consumer-driven contract tests. Adapted from [5]

4.1 Unit testing

Engine	Query	Cites
Google Scholar	microservice architecture pattern	[18, 19]
Google Scholar	microservice testing	[22]
IEEEXplore	microservice test automation	[20, 8]
Google Scholar	microservice test automation	[2]
IEEEXplore	automated testing of microservice architecture	[21]
Google Scholar	automated testing robustness	[9]

Table 1: Literature queries

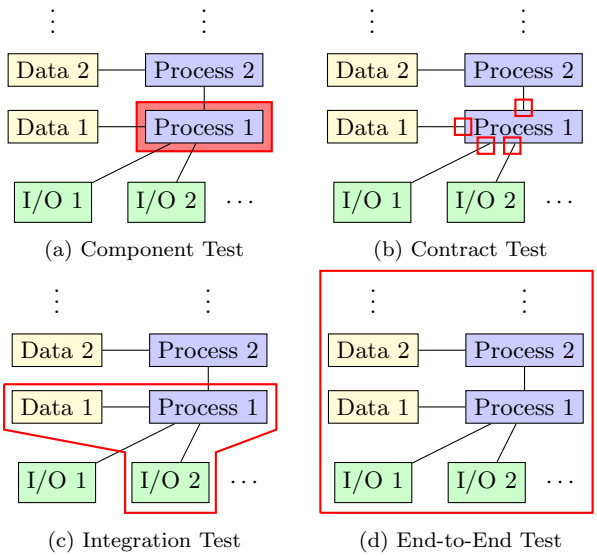


Figure 3: Exemplary extents for some Test Approaches

Unit testing is a method used to determine the efficiency of individual components within a software system, such as methods, modules or classes. It helps determine the effectiveness of these practicular units in order to establish their functionality as part of the larger software system. This process involves testing how these units behave with specific inputs and outputs without activating the entire system. Usually, unit tests are conducted by developers in temporary contexts, often employing drivers to simulate certain components. [17] Each programming language provides an approach to unit testing. For example, JUnit is a unit testing tool utilized in the context of Java.

4.2 Component Test

Component testing is a focused approach to test specific parts of the system in isolation, unlike an end-to-end stack trying to test the whole system. For isolation, this type of testing actively avoids interactions between the system components and often uses the internal code interfaces along with test doubles to replicate the dependencies. Component tests are usually faster to write, easier to maintain, and may not show problems stemming from the interactions between components. The effectiveness of a testing strategy is ensured by a fair balance of extensive component test coverage with fewer broad-stack tests. [4] The extent of this testing approach is visualised in Figure 3a.

4.3 Contract testing

Contract testing is a method of validating an integration point by independently testing each application to confirm that the message it sends or receives is in agreement with the contract that defines the shared expectations. It is designed to test the interfaces ("contracts") of services in microservices. It ensures that a service (the provider) will meet the expectations of its consumers. Consumers would define the interface in terms of defined inputs and expected results from the provider's API. These are then validated against the provider to ensure the provider's compliance. This technique helps in loose coupling between services and allows for early detection of integration issues, thus enabling independent development and deployment cycles. [16] There are plenty of tools available for contract testing and one of them is Pact. Pact is a code-first tool for testing integrations using contract testing. Pact offers advantages like independent execution and rapid feedback which ensures stability and simplifies maintenance. [15] The extent of this testing approach is visualised in Figure 3b.

4.4 Integration testing

Integration testing is an important step in ensuring the reliability of microservices architectures by focusing on how services interact and communicate with each other. It combines individual microservices to test their ability to work together seamlessly, supporting complete workflows and business processes. The primary goal is to detect problems like data mismatches, communication breakdowns, or integration failures that might be missed during unit testing.

In MS, this type of testing becomes important because of the decentralized nature of services and their dependence on APIs for communication. Integration tests confirm that services work together as intended in a larger system, resolving issues such as mismatched service contracts, incorrect data formats, or communication errors [2]

Integration testing comes with difficulties and challenges specially in managing dependencies and handling data consistency. To facilitate and solve those challenges mock objects are used. Mock objects or called as server stubs are used in testing by reproducing needed dependencies as fake objects, which have methods that return pre defined values. Additionally, to ensure proper interactions between the software component being tested and its dependencies, mock objects can be used to verify how the software component interacts with them. One of the frameworks used for mock objects is Mockito. It is popular for Java developement. It provides an easy way for developers to implement mock testing and offers an API to manage mocks. [13]. The usage of this kind of testing in MS is based on function calls. The extent of this approach is visualised in Figure 3c.

4.5 End-to-end testing (E2E)

End-to-end testing ensures that a complete and integrated system behaves as specified, especially in complex architectures like MS architectures. E2E evaluates the interaction of different components such as user interface, backend, database, and external integrations by simulating real-world scenarios. This testing method can detect integration issues, data inconsistencies, and workflow errors that might go unnoticed in other testing phases. Some methodologies like Behavior-Driven Development (BDD) and automated test scripts enhance its effectiveness. BDD focuses on user stories and collaboration among team members to ensure business requirements are met, while automated tests simulate user interactions and provide feedback. The benefits of E2E is the possibility of detecting integration problems early, reducing development defects, validating critical workflows, and improving system reliability and quality. [2] As for every testing method, some frameworks exist to make the testing process easier. Selenium is an open-source testing framework used for automating web applications and E2E. Developers can run tests with Selenium directly in various web browsers. Selenium provides tools for acceptance testing, regression testing. Selenium tests if MS-based applications meet specific requirements. [3] The extent of this testing approach is visualised in Figure 3d.

4.6 Automated black-box

Automated black-box testing for MSA enabled by Learning-Based Testing (LBT), combines machine learning techniques with model checking to support both functional accuracy and resilience assessments of distributed systems. This approach use incremental learning, where test cases are iteratively generated and assessed to refine a model representing the system under test (SUT). Model checking ensures that the system meets well-defined requirements, making LBT effective for high-throughput testing scenarios such as fault injection and comprehensive validation. By scaling test case generation and execution of complex distributed microservices, LBT addresses challenges like non-determinism and state management in distributed systems. Meinke and Nycander demonstrated this approach using the LBTest version 1.3.2 tool in their paper. [12]

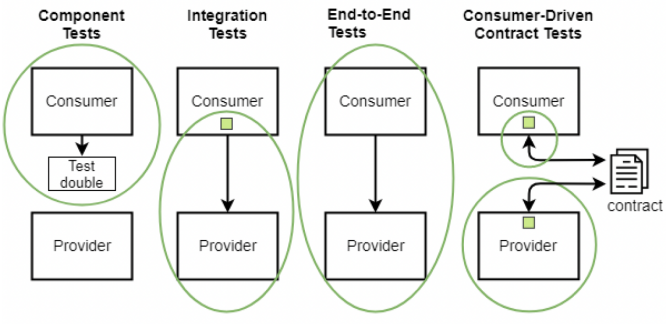


Figure 4: Scopes of different testing methods Adapted from [10]

Figure 4 illustrates different testing approaches in microservices, including component tests using test doubles, integration tests focusing on interactions between consumers and providers, end-to-end tests validating complete workflows, and consumer-driven contract tests ensuring compatibility between consumers and providers through explicitly defined contracts

Criterion	IEEE Definition [1]
Robustness	"The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions."
Reusability	"The degree to which a software module or other work product can be used in more than one computer program or software system."
Test Coverage	"The degree to which a given test or set of tests addresses all specified requirements for a given system or component."
Efficiency	"The degree to which a system or component performs its designated functions with minimum consumption of resources."

Table 2: Automated testing methodology criteria

5. COMPARATIVE ANALYSIS OF TESTING APPROACHES

All the testing approaches mentioned provide different coverage and applicability. From these differences the following research questions arise:

RQ1 How can each individual test automation approach be evaluated given predefined defined criteria?

RQ2 Are there challenges that apply to automated testing overall?

RQ3 Which testing strategies offer the best results at the lowest cost?

RQ4 Which areas can potentially be improved?

5.1 RQ1

Before any ranking can be performed, some criteria need to be set. Kropp et al. [9] discussed automated robustness testing. They describe robustness as mentioned by the IEEE [1]. This source also provides eligible definitions for other criteria. They are listed in Table 2. Since all of these criteria are hard to quantify in an absolute measurement, we rely solely on the options provided implicitly by the structure of each testing methodology.

Most of the demands stated by the four criteria are dependent on the implementation of both the MSA and the test mechanism involved which makes an absolute measurement of criterion-alignment impossible. It is therefore the difference in ease with which a developer can put testing mechanisms in place that are being evaluated. As performing proper surveys is out of scope for this work, we will construct a ranking for each criterion based on our perception of the testing approach and how well we think each paradigm enables conformance with each criterion. The overall ranking can be seen in Table 3.

Methodology	Robustness	Reusability	Test Coverage	Efficiency
End-To-End Testing	1	5	3	5
Integration Testing	5	4	1	4
Component Test	4	1	2	1
Automated Black-Box	3	3	5	3
Contract Testing	2	2	4	2

Table 3: Ranking each automated testing methodology for each criterion

5.1.1 Robustness

Robustness in the context of this work is primarily concerned with whether the automated testing implementations can handle a bigger and more complex structure without failing and whether they can manage errors induced by unplanned interface changes which might lead to unusual responses from individual services during the testing process.

End-To-End Testing.

End-to-end testing is concerned with the input and output of the entire system. Interface changes within the system and inter-service communication are therefore out of reach for the method and lead to errors within the testing algorithm less frequently. Errors cascading through the entire system can overwhelm End-to-End testing if not properly managed but the overall risk seems to be minimal compared to other testing methods as they are subject to more minor interface changes within the architecture.

Contract Testing.

Second in this ranking is contract testing. This method is more concerned with the interfaces of each individual service. While E2E testing only looks at the in- and output of services addressable by the user, every service can be considered here. This, of course, increases the overall risk to overwhelm the capability of the test algorithm to catch errors and unexpected behaviour. As this approach is only concerned with the interface specifications, each contact with a service is held shallow in comparison to other testing techniques, where an in-depth analysis of the algorithm could lead to more severe errors.

Automated Black-Box.

Automated black-box testing as an application of machine learning techniques and large data sets to apply to the MSA is more prone to failures in its expected behaviour as each new case introduces the possibility for unforeseen errors within the output handling. The main issue regarding robustness is the potential size of the test set. In combination with a potentially large system with many individual services the overall time and resources needed for testing might exceed the systems capabilities.

Component Testing.

Component testing is more concerned with the intricacies of a service. Using knowledge about the algorithm within a service, edge cases for its functions can be created and tested. Minor unnoticed changes to the input and return types of each function might therefore cause issues with the algorithm. This hazard is not present for the testing methods of high ranking and therefore introduces robustness issues if not monitored closely.

Integration Testing.

In last place of this list is integration testing. This paradigm tests groups of services within the system for their collective behaviour. The number of permutations that can be tested here and potentially hazardous interface changes similar to function declaration changes described for component testing might cascade to form outputs the testing system is not able to handle.

5.1.2 Reusability

Reusability is concerned with whether test cases, like input-output pairs and algorithms to check for conformance with predefined test cases, can be used multiple times in either the same test run for different parts of a system or in different test runs after for example changing a service implementation.

Component Testing.

When changing or upgrading certain services all component tests for all the other services still continue to work as they view each service in isolation. It is also possible that the cases for a changed service still remain applicable if the functions used stay the same regarding parameters and return values and functionality. Only when these aspects change then the tests need to be updated.

Contract Testing.

Contract tests are reusable in the sense that they apply to all services with the same contract and can therefore be used for prototype services aimed at replacing or improving upon already existing services with the same functionality. Similar to component testing, the existing tests for all non-involved services remain applicable regardless of the changes done to an altered service.

Automated Black-Box.

Automated black-box testing is applied to a specific service after training a machine learning algorithm. Small changes in the internal composition of the black-box can alter the results in unforeseen ways, which makes a retraining for an altered service necessary. This approach still retains the third highest ranking as, again, one change to a service does not alter tests for the others.

Integration Testing.

As integration is concerned with multiple services at once, changes to only a few services require changing cases and I/O for a larger number of cases. Changes to behaviour can cascade through the workflow of the test algorithm which invalidates existing cases.

End-To-End Testing.

End-to-end testing is often highly specific to a certain system. It is to some extend the extreme case of integration

testing where all services are tested in unison. The issue of integration testing, changes cascading through the behaviour of the entire system, is therefore only aggravated when monitoring the biggest possible set of interacting services.

5.1.3 Test Coverage

Each testing approach has other areas in which it excels regarding test coverage. In total the three main areas that have to be covered are correctness of each service, the communication between services and the system as a whole. A testing methodology scores higher if it covers more of these areas than another.

Integration Testing.

Integration testing offers the highest test coverage due to its versatility. While it does not check the fine grained mechanisms of each service, it is able to test any subset of services in unison, separating small work groups or larger areas regarding database management, user interaction, or processing. It also implicitly tests the communication between individual services during that process. As it exceeds in both communication and system coverage, it is ranked highest for this metric.

Component Testing.

Component testing acts in some sense as a complement to integration testing. As it does look at each service in detail individually, it covers cases which integration testing is not able to handle. As this method evaluates internal mechanisms of services it is able to offer great coverage on an atomic level of the MSA. Some communication is tested as component testing is also concerned with the I/O functions of a service.

End-To-End Testing.

End-to-end testing covers the functionality of the entire system. It does however not test individual services, subsets, or communication patterns. While its coverage can be argued to be the most important one, as it is closest to the real-world user interactions, this method still lacks in service and communication specific testing, leading to a lower score.

Contract Testing.

Contract testing is only concerned with the output of each individual service and therefore has the shallowest scope of all the methodologies compared. It is not concerned with the intricacies of an individual service, the system as a whole, and only parts of the communication mechanisms.

Automated Black-Box.

At last place is automated black-box testing. Cases generated by machine learning algorithms have no working knowledge of the algorithms used within the services and can therefore not reliably generate edge cases. Edge cases might have serious consequences for the user experience or the state of the system as a whole. Missing them therefore inevitably leads to bad coverage which is why this method is on the fifth place.

5.1.4 Efficiency

Efficiency in terms of minimizing resource consumption itself is again heavily reliant on the implementation for each individual case. We will therefore evaluate the efficiency of each testing approach by the amount of additional work that is to be expected. Running a test on the entire system will be more resource intensive then one that is performed on a single service after for example only said service has been altered.

Component Testing.

Component testing is the most efficient methodology. After performing changes to only one service, it can be tested individually for changed behaviour. Extensive testing of the entire system is more resource intensive but as this approach can cover remote test cases within each service, this additional resource consumption is well invested.

Contract Testing.

Contract testing is subject to the same benefits that component testing has. It can test remote edge cases on start up of a system but also re-test only select services when changes are to be rolled out. As it does not offer the same capabilities as component testing it places slightly behind.

Automated Black-Box.

Automated black-box can also be applied to select parts of the system. The biggest issue regarding efficiency is the potential need for a machine learning algorithm to be run which can be a resource intensive task. The test application on the other hand is fairly easy to execute once the cases have been generated and can be applied to the black-box.

Integration Testing.

Integration testing can involve certain services multiple times as part of different subsets of the entire system that are to be tested. While it does provide coverage for different applications for these cases, services are tested multiple times, requiring additional resources and the need to generate cases with them in mind. When changing one of the services, all subsystems that are a part of need to be retested in their entirety, increasing the amount of duplicate work even further.

End-To-End Testing.

End-to-end testing offers a good efficiency on startup of a system in comparison to other methods as the entire system needs to be tested either way. Any change made to a single component after that requires the entire system to be tested again, even if all of the other systems are not even involved with the changed one. This results in a bad efficiency over time, leading to last place on this ranking.

5.2 RQ2

All testing methods share some common issues regarding the criteria used for evaluation. Robustness is in general at risk when introducing large systems, especially earlier in development, when quickly changing interface and function specifications can result in datatype mismatches, or exceedingly large loads placed onto the testing infrastructure. Another bigger issue regards the combination of efficiency and test coverage. An in-depths analysis of a system requires large sets of test cases for many different levels of operation

Methodology	E	Rank
End-To-End Testing	2.4	5
Integration Testing	3.3	3
Component Test	15	1
Automated Black-Box	2.7	4
Contract Testing	6	2

Table 4: Effectiveness Ranking of each Testing Methodology

from individual functions within services to the whole system as a whole. Automating this process inevitably leads in duplicate work, securing better test coverage at the cost of efficiency.

5.3 RQ3

As this ranking is only subject to our own opinions, so is this summary of all the ranking scores. To answer this question, we first generate the combined score of robustness, reusability, and test coverage as these are the metrics that constitute what we called "best results". We define a higher score as better which is why we first invert the values of robustness rob, reusability reu, and test coverage tes. We do that by subtracting each value from the number 6. We then divide the combined score by the efficiency value eff to generate one combined value we call "effectiveness" E.

In our opinion not all of the ranks proposed are equally important. We think, that the most important aspect for a test methodology is the test coverage as a test that does not cover enough cases or enough parts of a system is not as useful even if it is for example robust. We therefore attach a weight of 2 to that value. The remaining scores keep a weight of one. Combined we generate equation 1.

$$E = \frac{2 \cdot (6 - tes) + 1 \cdot (6 - rob) + 1 \cdot (6 - reu)}{eff} \quad (1)$$

The results then imply an overall ranking which is displayed in Table 4. Based on this evaluation, component tests have the greatest effectiveness and E2E tests the lowest.

5.4 RQ4

It is our opinion that no single test methodology can satisfy an in depth test of a system, especially not when automated. It would therefore be reasonable to combine multiple approaches and only use them when necessary. This is already a common approach. The study conducted by Wasseem et al. [21] marked 33 articles as its primary studies but listed 79 individual applications of certain testing methods in those studies. Therefore multiple methods had to be used per study. Determining when a set of tests is to be run is a question that would need to be answered to minimize redundant work and improve upon resource usage. A component test is for example mostly applicable when changing a single service fundamentally. A full end-to-end test run might be applicable too, but such an optimisation would need to be made per system and cannot be generally decided.

6. THREATS TO VALIDITY

This work has been conducted with the methodology described above and the papers we considered useful for our evaluation. It is unlikely that we provide a complete overview since we only used two search engines. Our prompts too could miss important papers which may not use any of the keywords we were looking for, for example if a paper reported on architectures resembling MSAs without mentioning them explicitly. We do not provide a description of all architectures mentioned in the papers due to a low usage rate or only nuanced differences between them.

The answers given to our research questions are primarily based on our own perception of the status quo of automated MSA testing. A more objective approach like conducting surveys with MSA developers might yield different results closer to the reality of this subject.

7. CONCLUSION

In this study, we analysed a set of automated testing methodologies for microservice architectures regarding their efficacy in terms of robustness, test coverage, reusability, and efficiency. The results show that E2E testing is the best in terms of robustness and good in test coverage because it verifies the functionality of the whole system, but it still remains resource intensive. Integration testing is superior in Test coverage of a system by looking at the interaction between services, while component testing provides reusability and allows for in-depth validation of each service. Contract testing focuses on the evaluation of service interfaces, ensuring loose coupling between services. Automated black-box testing shows scalability potential but faces some challenges. We have also seen that multiple test methods are being used in unison to grant better coverage. One of our proposals was to find a way to properly evaluate whether a test run of a given method and test set is applicable after a change was made and to therefore save resources.

Microservices has shown that they offer a strong architecture based on resilience and scalability. However, Microservices are still a complex system which comes with constraints and challenges. Artificial intelligence and machine learning will play an important role in evolving the testing process of MSA systems by generating automated test cases, identifying edge cases, and predicting possible failures of large-scale distributed systems. In our opinion artificial intelligence will help facing some of the challenges discussed in this paper, deliver a better test experience for the user, and make automation processes more easily adaptable. Artificial intelligence also can help in the setup, orchestration and optimization will result reduce testing time, improve test coverage and improve the general reliability of MSA systems. An article at Software testing magazine supports our opnion by highlighting the role of artificial intelligence and machine learning in software testing. [11]

We believe that, in the future, a hybrid approach will be essential in combining the strengths of multiple testing paradigms. The combination of End-to-End testing for overall system validation, integration testing for inter-service communication, and component or contract testing for service-level accuracy will give developers and testers a better and more efficient testing strategy.

For a more accurate representation of the status quo in MSA development we propose a repetition of this work using surveys conducted with MSA developers to better represent real world perceptions of the different test methodologies presented.

8. REFERENCES

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

[2] V. R. R. Alluri, P. Katari, S. Thota, S. G. Reddy, and A. K. P. Venkata. Automated testing strategies for microservices: A devops approach. *Distributed Learning and Broad Applications in Scientific Research*, 4:101–121, 2018.

[3] A. Bruns, A. Kornstädt, and D. Wichmann. Web application tests with selenium. *IEEE Software*, 26(5):88–91, 2009.

[4] T. Clemson. Testing strategies in a microservice architecture. `https://martinfowler.com/articles/microservice-testing/`, 2014. Accessed 9.12.2024.

[5] M. Cohn. *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Professional, 1st edition, 2009.

[6] E. Dustin, J. Rashka, and J. Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley Professional, Boston, MA, 2009. Referenced pages 3–14.

[7] A. Elsayed, T. Cerny, J. Y. Salazar, A. Lehman, J. Hunter, A. Bickham, and D. Taibi. End-to-end test coverage metrics in microservice systems: An automated approach, 2023.

[8] M. J. Kargar and A. Hanifizade. Automation of regression test in microservice architecture. In *2018 4th International Conference on Web Research (ICWR)*, pages 133–137, 2018.

[9] N. Kropp, P. Koopman, and D. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)*, pages 230–239, 1998.

[10] J. Lehvä, N. Mäkitalo, and T. Mikkonen. Consumer-driven contract tests for microservices: A case study. In X. Franch, T. Männistö, and S. Martínez-Fernández, editors, *Product-Focused Software Process Improvement: 20th International Conference, PROFES 2019, Barcelona, Spain, November 27–29, 2019, Proceedings*, volume 11915 of *Lecture Notes in Computer Science*, pages 497–512. Springer Nature Switzerland, 2019.

[11] S. T. Magazine. The role of ai and machine learning in modern software testing. 2025. Accessed: 2025-01-15.

[12] K. Meinke and P. Nycander. Learning-based testing of distributed microservice architectures: Correctness and fault injection. In *Lecture Notes in Computer Science*, pages 1–12, 2015.

[13] S. Mostafa and X. Wang. An empirical study on the usage of mocking frameworks in software testing. In *2014 14th International Conference on Quality Software*, San Antonio, Texas, USA, 2014. IEEE.

[14] S. Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., Sebastopol, CA, 2015.

[15] Pact Team. Pact documentation, 2025. Accessed: 2025-01-09.

[16] I. Robinson. Consumer-driven contracts: A service evolution pattern. `https://martinfowler.com/articles/consumerDrivenContracts.html`, 2018. Accessed 9.12.2024.

[17] P. Runeson. A survey of unit testing practices. *IEEE Software*, 23(4):22–29, 2006.

[18] D. Taibi, V. Lenarduzzi, and C. Pahl. Architectural patterns for microservices: A systematic mapping study. *Closer*, pages 221–232, 2018.

[19] J. A. Valdivia, A. Lora-González, X. Limón, K. Cortes-Verdin, and J. O. Ocharán-Hernández. Patterns related to microservice architecture: a multivocal literature review. *Programming and Computer Software*, 46:594–608, 2020.

[20] Y. WANG, L. CHENG, and X. SUN. Design and research of microservice application automation testing framework. In *2019 International Conference on Information Technology and Computer Application (ITCA)*, pages 257–260, 2019.

[21] M. Waseem, P. Liang, G. Márquez, and A. D. Salle. Testing microservices architecture-based applications: A systematic mapping study. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, pages 119–128, 2020.

[22] M. Waseem, P. Liang, M. Shahin, A. Di Salle, and G. Márquez. Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software*, 182:111061, 2021.

Microservices Architecture

Insights into Benefits, Challenges, and Best Practices

Rares Eugen Marc
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
rares.marc@rwth-aachen.de

Muhammad Omar
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
muhammad.omar@rwth-aachen.de

ABSTRACT

Evolving from the ideas of service-oriented computing, the microservices architecture has gained significant traction in software engineering over the past decade and is now a widely adopted approach among practitioners and researchers. Many organizations are now choosing to build software by decomposing it into small, independent services, each in a business-bounded context, created and managed by small autonomous teams. This paper explores the benefits and challenges associated with adopting microservices, emphasizing some key technical and organizational considerations.

Through a comprehensive review of existing literature and case studies, we identify key aspects. These include technical considerations of microservices – scalability, reliability, maintainability, and security, as well as organizational impacts – on team management, knowledge transfer, and rapid iteration. We discuss these aspects individually, and go into detail about the opportunities and challenges they present. Then we look into how these aspects interrelate, highlighting how the advantages of one can lead to challenges in another.

Finally, we provide practical advice for practitioners and organizations considering the transition to microservices, giving heuristics for determining when microservices are a good fit, and when they may not be. This discussion aims to equip software engineers and organizational leaders with the necessary insights to effectively navigate the complexities of the microservices architecture in full-scale software engineering.

Keywords

Microservices, Microservice Architecture, Software Architecture

1. INTRODUCTION

The term "microservices" was popularized by James Lewis and Martin Fowler in their 2014 article [12], where they describe microservices as an approach to developing a single application in terms of a collection of small, independent services. Each service runs in its own process, and communicates with other services using lightweight mechanisms like HTTP requests. These services are organized around business capabilities and can be deployed through Continuous Integration and Continuous Delivery/Deployment pipelines, which reduces the need for centralized management of services [12].

The shift from monolithic to microservices architecture stemmed from the limitations of scaling and maintaining large, single-unit applications. In a monolithic structure, all components like the user interface, server logic, and database are tightly integrated, making the entire system dependent on atomic updates and deployments. While monolithic systems are straightforward and manageable when small, they become difficult to scale and modify as they grow [2]. For instance, a minor change requires redeploying the entire system, and scaling demands the duplication of all components rather than specific, high-demand areas. These constraints made it challenging to adapt to the needs of cloud deployment and agile development cycles, pushing many organizations to explore microservices [12].

Microservices offered a new approach by splitting applications into loosely coupled, independently deployable services. This structure allows teams to scale, deploy, and maintain individual services more easily, even using different programming languages where beneficial. However, over time, microservices have presented new challenges, especially around managing inter-service communication, data consistency, and increased complexity [5]. Engineers now face issues like the overhead of managing numerous services, the challenge of ensuring reliable communication, and troubleshooting failures across a distributed system. These factors are leading some teams to reconsider aspects of microservices or explore hybrid approaches that combine the simplicity of monolithic structures with the flexibility of microservices [10].

In this paper, we explore the benefits and challenges of adopting the microservices architecture as the core principle of building large-scale software systems, focusing on both technical and operational considerations. The paper is structured as follows: Section 2 describes the Research Methodology used to identify key aspects of microservices, drawn from the perspectives of both practitioners and researchers. Sections 3 and 4 detail the Technical Considerations and Organizational Impacts of implementing microservices architecture in large-scale software engineering. In Section 5 we provide a Discussion, offering insights and advice rele-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2024/25 RWTH Aachen University, Germany.

vant to practitioners who are considering the adoption of microservices. Finally, Section 6 presents the Conclusion and gives some potential directions for future work.

2. RESEARCH METHODOLOGY

In this section, we outline the research methodology we used to investigate the advantages and disadvantages of microservices. We conducted a comprehensive literature review, focusing on both technical and organizational aspects. Key areas of investigation include scalability, reliability, maintainability, security, team management, skill diversity, and rapid iteration. This multi-faceted approach should provide a more nuanced understanding of the impact of microservices on software systems and organizations.

2.1 Research Process and Approach

In order to effectively research and present the advantages and disadvantages of microservices, we first needed to establish the key areas of the topic, namely the aspects of a system which are the most impacted by this type of architecture.

We began by trying to understand how such a system works and the differences to a classical monolith, so we thoroughly reviewed articles focusing on this topic, starting with Fowler's 2014 blog post [12]. Next were some success stories of companies making the transition by way of necessity, not a voluntary endeavor in order for their systems to fit into this category. Such an example is Amazon [8], who made this change naturally in order to adapt to their rapidly increasing customer base, which also prompted them to work based on the motto "Get big fast".

Next, we looked into the peer-reviewed literature around microservices, focusing on keywords that allowed us insight into the practice, such as "design and architecture", "challenges", "pains and gains", "bad smells", and "trade-offs". We used IEEE Xplore and ACM Digital Library to find these papers, using the built in search tool to filter papers.

Among the papers we found, Jamshidi et al. provided a particularly comprehensive overview in their paper "Microservices: The Journey So Far and Challenges Ahead" [10]. In another paper focused on the industry perspective, Bogner et al. [3] provided the results of a survey conducted among software professionals. Their focus on the perceived characteristics of microservices among experts [Figure 1] helped us draft an outline of the most interesting aspects.

The promises presented in the literature and the needs of the industry were clearly aligned, so in order to verify our findings, we looked into systematic literature reviews, such as the one by Vural [22], to confirm our hypothesis using actual data. "A Survey on the State of Practice" by Ghofrani and Lübke [7] also offered us some insight from real practitioners working in the field who already have a lot of experience with the topic. They presented their challenges and their reasoning when working with microservices, and they summarized their findings in a bar graph [Figure 2].

Based on these steps, we were able to determine that simply a technical approach to the topic would not suffice, and in order to have enough perspective, a non-technical view is also necessary. Because such a system functions in a very different way, the organizational impact it has on the design, development, and management process cannot be glossed over. Thus, we adjusted our approach when reviewing the literature and extracted the fundamentals from both aspects.

Figure 1: Impact of Microservices on Software Quality, adapted from [3]

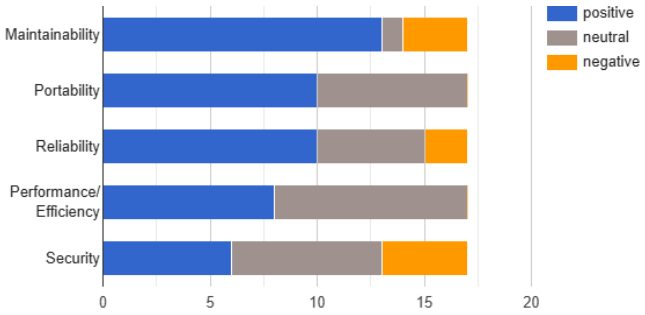
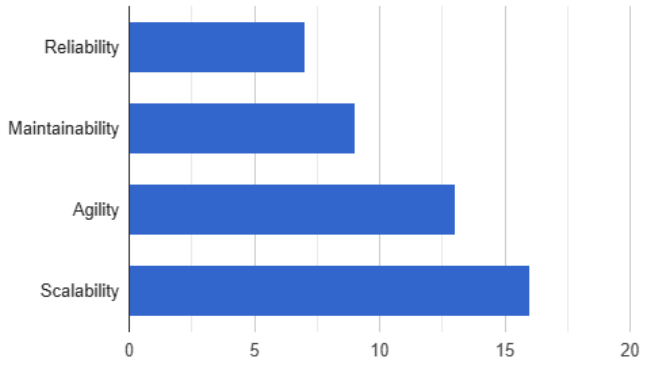


Figure 2: Goals when deciding on Microservices architecture, adapted from [7]



2.2 Key Aspects

Scalability is decidedly at the forefront of technical considerations, being of paramount importance in almost all cases. Ghofrani and Lubke's survey placed it as easily the most wide spread end-goal of people who adapt this architecture. Since availability is a priority for most enterprises, it is no surprise that **reliability** is also significant.

In their study involving software professionals from Germany [3], Bogner et al. tried to determine the impact on different software qualities by microservices; **maintainability** came first, since the motivation to reuse and structure services around business capabilities is the driving factor in the design phase of such architectures. **Security** is a more divisive topic, but because of this reason and the fact that we believe it to be an aspect that must always be considered, it is also included in our paper.

Regarding organizational matters, microservices have had a significant influence on **team management** and structure, as well as their distribution and style of work; challenges in **skill** and **tool diversity** have therefore emerged as a consequence. On the other hand, there are clear advantages of these changes which reflect in a **rapid iteration**. The importance of these aspects are confirmed in the aforementioned studies through the performance, efficiency and agility metrics.

The decision to concentrate on these features is, therefore, grounded in the emphasis placed on them in existing literature, as well as their relevance in real-life contexts. This

dual-faceted view on microservices informs the reader on practical benefits but also organizational complexities involved in adopting this architecture.

3. TECHNICAL ASPECTS

Microservices architecture involves breaking down applications into smaller, independently deployable services. This approach offers benefits like scalability, fault isolation, and flexible technology choices. However, it also introduces challenges such as increased complexity, distributed data management, security concerns, and service orchestration. In this section, we look into the scalability, reliability, maintainability, and security of microservice-based systems.

3.1 Scalability

The term "scalability" is defined as the property of a system which defines to which degree it can handle growing or shrinking workloads or adapt its capacity to handle variability [1].

Independent deployment is one of the biggest features thrown around, which does indeed have a myriad of advantages. Since all services act as self-contained units, updates or fixes to one of them do not require redeployment of the entire system. This also allows for a more precise scaling of only the components that are actually facing problems. All this can be done at runtime, since taking down one service does not affect the whole application, leading to an efficient use of resources and rapid reactions to changes.

"Smart endpoints, dumb pipes" - the colloquial term for another benefit of this separation, named this way because of the following behavior: microservices receive a request, apply specific logic on it and forward a response - they behave similarly to filters, which means there is very little need for central control.

Different environments mean that developers are also free to choose different databases depending on what is best suited in each case, therefore obtaining significantly more efficient storage. Not only that, but since services operate within a bounded context, they can be safely worked on without needing to be familiar with the rest of the application's architecture.

Cloud services have also become prevalent in dealing with microservices; they can handle features common to most components, such as managing servers, data stores and handling errors. This innovation combined with independent deployment made people recognize how well-suited the architecture is for cloud development; it can take the most advantage from the elasticity and on-demand characteristics of such services. An argument can also be made for the ability to redeploy a service in another location in order to take advantage of network positioning.

Drawbacks do exist when working with an interconnected web of completely separate services, such as a larger consumption of network and computing resources. Remote calls are much more expensive than in-process calls used in a monolithic application, therefore they need to be carefully designed in order to be more coarse-grained and avoid chatty communications. A different runtime environment also needs to be maintained for each service.

Monitoring a dynamic network of microservices involves difficulties: constantly having an overview of the whole system can become quite challenging. The flow between an ever-increasing number of units can become hard to control and update, and track of all versions and compatibilities between them must be kept. There is also difficulty when performance testing - measuring the performance of each service and also of the overall application can get complicated.

Duplicate effort is also an unfortunate consequence of separate infrastructures, since many similar issues will appear in different microservices. The effort wasted on managing infrastructure is what led Amazon to transition to the aforementioned cloud services.

Data management is an issue which cannot currently be marked as resolved. Managing updates on multiple servers does not have a clear solution, since distributed transactions are not only hard to implement, but their key two-phase commit feature damages the throughput of the application due to the overhead introduced. The current consensus is settling for eventual consistency, but this does not apply when dealing with sensitive data. Another difficulty stems from the previously mentioned possibility to use different databases; creating queries which combine data from distributed and heterogeneous stores can be challenging.

3.2 Reliability

Reliability is achieved in microservices through a combination of certain design principles and mechanisms that maintain functionality even under failure conditions.

By design, the system is **fault tolerant**; flexibly adding or removing services is possible without affecting the entire system, since most of them can work independently, even if others are not deployed. This also means that updates can be made while the system is running, enabling the continuous delivery of improvements.

Bounded contexts is another factor which aids greatly in isolating faults and addressing failures without affecting anything else. Services operate within their own separate runtime environment, adding another layer of isolation and preventing issues caused by shared resources. They are also designed following the loose coupling principle, which ensures that dependencies are minimized, hence reducing the risk of cascading problems.

The shotgun method is a practice which distributes a single point of failure into multiple, essentially providing many smaller instances of services which can quickly take on the workload of one that failed [15]. Netflix leans heavily on this method. Here also intervene load balancing and service discovery mechanisms, which can use health checks to find overloaded services and rebalance traffic by redirecting it to healthy instances.

Containerization is leveraged to great a extent by this architecture, making use of lightweight environments that ensure quick and independent failure. Most services are therefore designed to be stateless, in order for other instances to be able to quickly take on their traffic or to quickly be started and deployed again.

Some of these benefits can however be undermined if the challenges they introduce are not addressed properly.

Cascading failures is an issue which appears when microservices are not properly isolated; the failure of one service causes another one to fail due to the dependency between them, and so on. This usually happens when they are too tightly coupled or if they have shared resources; thus why a decoupled design is of paramount importance. The boundaries of each service must also be well-defined, since

badly designed interfaces can lead to increased communication and therefore worse performance and more instability.

Communication issues are bound to occur more often than in other architectures since a much larger number of messages are being sent. On top of this, each service has its own network connection, therefore problems with latency and connectivity are more likely to appear.

These issues can however be addressed using different monitoring mechanisms which detect failures and prevent them from spreading. An example is circuit breakers; they are employed to ensure that, once a service starts failing, it stops receiving requests and is permitted to slowly recover until it becomes functioning again. The purpose is to allow services to degrade gracefully instead of having them break totally.

Debugging is an activity heavily affected by this type of architecture; the large number of microservices are operating independently of each other and are therefore also writing their own logs. Delving through distributed, extensive logs is a difficult endeavor and makes it significantly harder to identify problems.

3.3 Maintainability

Maintainability is often cited as the main migration driver behind microservices [3], owing to its modularity by design and bounded contexts.

Enforcing **firm module boundaries** makes it easier to respect service separation and improves code modularity. By organizing software systems around business capabilities rather than technical layers, teams can more easily isolate services, enabling more focused updates and clearer ownership. This organization often aligns with Conway's Law [4], where the system's structure mirrors the organization's communication and workflow patterns, thereby fostering maintainability. Despite these advantages, microservices also introduce a risk of complexity in the form of an interconnected death star dependency graph [14], where services depend on one another in intricate, sometimes fragile ways.

However, **limitations in service decomposition** may arise in certain contexts. While microservices promise flexibility in design, not all software systems are easily split into discrete services. Certain applications may not naturally decompose along business capability lines, leading to difficult decisions about service boundaries. This issue is compounded by the need to continually refactor and redesign services to ensure they remain maintainable and aligned with evolving business goals [20]. This means that microservices are not a one-size-fits-all solution and significant effort might be needed to ensure maintainability as the system scales.

Communication challenges scale when more services are added, increasing the complexity of communication between microservices. Each connection introduces potential failure points and adds complexity. Instead of being eliminated, the inherent complexity of the domain might be shifted to inter-component communication to simplify system components. [12]. The challenge of maintaining coordination can be mitigated by shared API, protocols, and design principles. Clear inter-team communication via thorough documentation and through mechanisms like scrum of scrums [16] also help to align multiple teams as they work on different services with different release cycles.

Maintaining **non-functional qualities**, such as scalability, security, and performance, is also closely tied to the overall maintainability of a microservices architecture. The decentralized nature of teams and services can create gaps in responsibility unless strong oversight and platform-level teams are in place to manage cross-cutting concerns. [5] These teams help ensure that non-functional qualities are consistently maintained across all services, reducing the risk of the system becoming fragmented or unreliable. If these qualities are not actively managed, the system may face increasing maintenance overhead as new issues arise post-deployment.

3.4 Security

Security is not a core consideration in the comparison between traditional and microservices architectures, not being heavily affected by the differences, but it is nonetheless an important topic.

Finer-grained security policies represent the main advantage here; policies can simply be tailored to each and every microservice to best suit its usage. This means very detailed access controls and configurations specific to each service's needs, which allows a better application of the least privilege principle than in a monolith. On the other hand, automated security policies are also possible; developers can create simple, abstract measures and apply them to multiple services with similar reasonings regarding security. More important services with sensitive data can be designed to have stricter access or dedicated measures such as encrypted communication, token-based authentication and others. To this end, firewalls can also be configured at multiple layers: in API gateways, service meshes or individual containers.

Isolation is also significantly beneficial, since a compromise is less likely to spread across the system. A vulnerability which is exploited in one service does not mean that the attacker can easily use the resources and data of another.

The challenges in this architecture are however also quite significant.

Provenance and Authenticity of requests must quickly and correctly be established at the level of each microservice. This poses a few problems both for decentralized and centralized access control systems; decentralization means maintaining consistent authentication mechanisms across all services, which becomes difficult due to each of them evolving at a different pace. Centralization however can easily lead to bottlenecks and slow response times. On this topic, due to the trust placed between services, if an attacker manages to compromise one of them, he can then send malicious requests to others and therefore extend the intrusion to an entire cluster of services.

Updating authenticating information also presents a challenge whenever a new service or user is added, since either a central authority must be kept up to date on every change, becoming a maintenance burden, or individual services must be updated consistently and at the same time, which is error-prone and may lead to mismatched or outdated credentials.

The attack surface in microservices is significantly larger, directly increasing the risk of intrusions. Endpoint proliferation is one of the causes; each service needs to open new ports and expose more APIs, creating more possible entry points for attackers. Another reason is that communication between services often happens over a network, as opposed to in-process calls of monolithic architectures, and messages are therefore exposed to potential interception, tampering or unauthorized access.

4. ORGANIZATIONAL ASPECTS

While the technical aspects of the microservices are often the main topic of discussion, the organizational challenges of moving to a microservices architecture must not be underestimated. We consider the aspects of team management, knowledge continuity, tool diversity, and rapid iteration and discuss their impact on an effective microservices implementation.

4.1 Team Management

"You build it, you run it", the principle introduced by Amazon's CTO Werner Vogels in 2006 [8] is central to the microservices architecture. It encourages small, decentralized, cross-functional teams that assume end-to-end ownership of their services. This aligns with Conway's Law, which says systems are shaped by the communication structures of organizations [4]. Such teams align with the principles of domain-driven design and bounded contexts [5], promoting productivity and allowing teams to make independent technology choices and deploy changes without waiting for others.

Coordination challenges may pose a significant problem due to the decentralized nature of microservices teams. Misalignment across teams can occur without strong workflows and effective communication, causing microservices to risk not achieving their intended benefits and instead introduce bottlenecks. [21]. Teams need more sophisticated workflows for inter-service communication and data management. To reduce overhead, organizations must foster a culture of robust DevOps, documentation, and knowledge sharing. [5].

Structured workflows and regular communication help to streamline processes and prevent inefficiencies. Establishing cross-team communication, thorough documentation, and standardized practices can help alleviate these coordination challenges, allowing teams to operate independently while supporting organizational goals [11]. Continuous integration and shared tooling should enable a more successful microservices implementation [3], creating a balanced ecosystem where independence does not come at the expense of fragmentation.

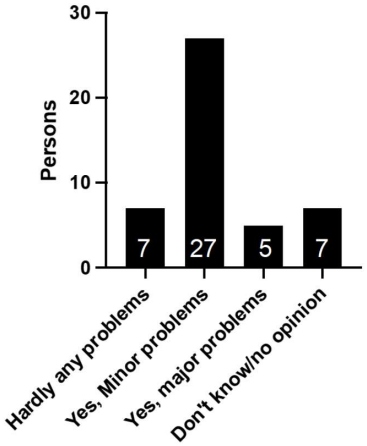
Maintaining system cohesion is a significant challenge due to the high level of team autonomy. Standardization and oversight are important, otherwise teams risk creating redundant features or incompatible solutions, which can fragment the system and complicate maintenance. Enforcing shared APIs, protocols, and design principles helps ensure interoperability across services, while a centralized platform engineering team can manage cross-cutting concerns like authentication, logging, and monitoring.

Technical standardization via solutions like shared APIs and service mesh frameworks can prove valuable for maintaining cohesion, as they simplify inter-service communication and enhance observability. Service meshes ensure consistent routing, load balancing, and security protocols across services, which is particularly beneficial in maintaining seamless integration across autonomous teams [18]. These strategies, combined with a culture of proactive communication and shared knowledge, ensure that teams retain their autonomy while maintaining a cohesive and reliable system.

4.2 Tool Diversity and Knowledge Transfer

Tool proliferation can increase across services, as multiple solutions may address similar needs in diverse ways. This can cause redundancies and make it challenging for developers to adapt to new tools and frameworks when switching services [17]. In a survey of practitioners [19], a majority of experienced software engineers expected at least minor problems stemming from increasing tool diversity [Figure 3]. Standardizing a subset of technologies and enforcing robust documentation practices can help mitigate these drawbacks, enabling teams to leverage the adaptability of microservices without overwhelming them with unnecessary complexity. Newman notes that organizations employing these best practices achieve smoother transitions between teams and better maintainability of services [13].

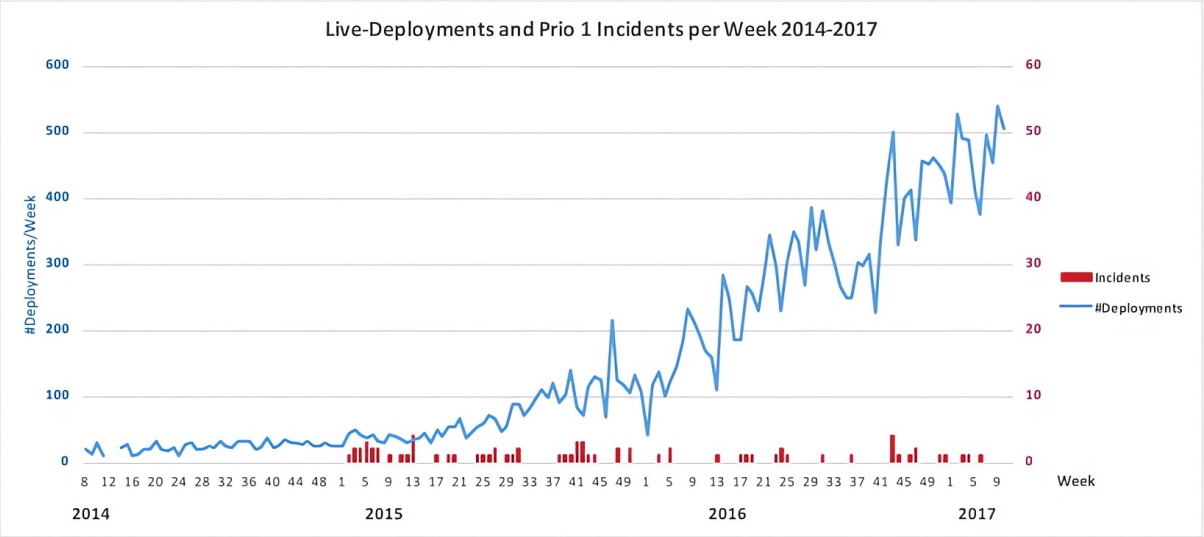
Figure 3: Perceived number of problems caused by increasing tool diversity, in the eyes of practitioners [19]



Skill diversity also introduces a challenge in microservices, as teams require broader skill sets. Each team takes on full responsibility for its service's life cycle, so team members must possess broader skill sets, including knowledge of development, operations, and possibly multiple technology stacks. This requirement can complicate recruitment and increase the need for cross-functional training, leading to a higher cognitive load for team members. While small, agile teams are generally efficient, having to put on multiple hats may stretch the individual members, which can impact overall efficiency [3].

Knowledge sharing and cross-team communication are essential to ensure system cohesion and mitigate the cognitive burden on team members [11]. As services may often be added, modified, deleted, and may change ownership, knowledge continuity via comprehensive documentation and inter-team communication are necessary. When team members switch roles or new teams assume responsibility for existing services, inconsistent or missing documentation can impede knowledge transfer, leading to delays and reduced productivity. Bogner et al. note that organizations successful with microservices tend to allocate resources to training, documentation, and shared tooling [3]

Figure 4: Number of live deployments and incidents per week at otto.de from 2014 to 2017. [9]



4.3 Rapid Iteration

Companies and projects that adopt microservices frequently report **reduced lead times** and increased release frequency [3], as the microservices architecture accelerates development speed by enabling granular and independent updates to individual services, allowing rapid innovation and adaptability. Each service is designed to perform specific functions, allowing development teams to release changes, address bugs, or even rewrite the service without disrupting the broader system.

In a 2017 case study, Hasselbring et al. [9] looked into the E-Commerce company otto.de and found out that since adopting the microservices architecture, the number of live deployments they made per week rose from 40 to 500 [Figure 4]. Notably, there was no significant increase in live incidents, which stayed at a relatively low level. They ensured reliability and quality via **automation of DevOps and testing pipelines**, which increased agility and facilitated faster turnaround times.

The independence of services allows teams to adopt **individualized workflows** and specialized languages or frameworks that best suit each service's requirements. As a result, organizations can respond quickly to market demands and introduce new features, contributing to a continuous deployment model that supports rapid iteration [12].

5. DISCUSSION

The microservices architecture presents a compelling proposition for anyone looking to develop a modern software system. By breaking down monolithic applications into smaller, independent services, organizations can produce more scalable, reliable, and maintainable software, while achieving greater agility and innovation.

However, the implementation of microservices does not come without costs. The complexity of managing a distributed system, the overhead of inter-service communication, and the potential for cascading failures must be taken into consideration, just to name a few. Organizations have to carefully weigh the benefits against the costs before embarking on a microservices journey.

5.1 Challenges in Practice

Bogner et al. [3] noted in their survey of microservice-based systems that, in practice, such systems are likely to fall short of their intended architectural goals. DevOps and automation practices were found to be rather mediocre in the systems that they assessed, and only very few companies strictly followed the "you build it, you run it" principle. They also found a lack of the high degree of technological diversity that is commonly expected with microservices, which shows a disconnect between what teams may plan to achieve from microservices, and what is actually built at the end of the day. Microservices certainly are not the silver bullet that some advocates may make it seem.

When considering microservices, every decision must be carefully weighed, keeping in mind that one step forward in one direction can mean two steps backward in another. By trying to increase the quality of one aspect of our application, another one may be affected for the worse. In terms of scalability, independent deployment offers great benefits, but it demands communication through API calls; this proves to be a disadvantage for the security of the application, since it now introduces endpoint proliferation and authentication challenges. Another example is the relationship between security and reliability; bounded contexts mean better isolation of faults and a decrease in risk of cascading failures. However, more effort needs to now be placed in determining the provenance and authenticity of requests to ensure safe calls - which in turn can introduce more communication issues negatively impacting reliability.

The organizational impact must not be underestimated as well. The microservices approach to building applications creates new types of challenges and requires new technologies, such as service discovery, workload balancing, container management, and strict DevOps and automation practices. These all introduce a significant amount of complexity which needs to be handled by a proportionate amount of expertise, which may or may not be available on the teams responsible, due to a lack of training or an increase in cognitive load. More coordination and communication is also required, since each team is now almost oblivious of anything that is being

developed outside the scope of their respective services.

5.2 Recommendations

Microservices must be therefore be used on a case-by-case basis, which of course prompts the question of how to recognize the appropriate situations. We have managed to extract some reliable practices from the literature and also draw some conclusions:

Do not start with microservices Fowler gives us some insight into this [6] where he cautions against the deliberate use of this architecture before it is actually necessary. Only when the downsides of a monolith application become too severe to handle should a transition be considered on a case by case basis. Well defined module boundaries also offer some of the benefits that people look for in microservices; this should rather be the first recourse.

Take full advantage of the appropriate tools DevOps, testing automation, and cloud services are also essential for an optimal use of microservices; distributed systems are notoriously difficult, and the complexity introduced by the new technologies and maintenance tasks required needs to be handled properly with a healthy measure of automation.

Consider the services' size In short: avoid making microservices too small. Even though services are ideally modeled around business capabilities, but this does not always yield optimal separation. Going too "micro" can lead to large overhead in infrastructure management and communication between services, which can offset the advantages gained through this architecture. Services should be split when they start having issues similar to monolithic architectures, such as having difficulties scaling or suffering from slow deployment.

Plan for organizational changes Microservices introduce significant organizational challenges, including increased complexity, the need for specialized skills, and heightened coordination demands, which should be taken into account when planning to move to this architecture.

In "The Journey So Far" by Jamshidi [10], he identified a problem in the current literature on microservices: there is a significant number of studies, but there has been very little impact on the common practice. He posits that the reason for this is researchers' lack of access to industry-scale applications using microservices.

In our research we have discovered a number of papers presenting studies that include industry experts, which do shed some light on the actual state of practice and alignment with the literature. There is not enough, however, for us to take this as matter of fact and declare that the two are perfectly in sync; it is entirely possible that there continues to be a gap between the two. There is likely still much room to evolve, but in order to do so, a good amount of coordinated effort between the two sides may be needed until academia can continue finding solutions for some of the issues identified here.

6. CONCLUSION

Over the years, the microservices architecture naturally evolved from prior ideas like the Service-Oriented Architecture (SOA) [23]. Popularized due to its promised benefits in scalability, reliability, maintainability, and team management, it has been widely adopted by the software industry to build large-scale software systems. In this paper we reviewed existing literature and experiences from practitioners to provide an overview of the advantages, challenges, and considerations associated with the adoption of microservices.

Our findings reveal that while microservices enhance system flexibility, maintainability, and independent scaling of components, they also introduce complexities in inter-service communication, data management, and system monitoring. There is also significant organizational impact, often leading to restructured teams aligned with business capabilities, which can improve agility but may present coordination challenges.

The relationship between different aspects of the microservices architecture is complex. The scalability benefits may lead to increased complexity in maintaining system-wide security policies, while the improved fault tolerance can complicate debugging processes across distributed services. Hence careful consideration is important when adopting microservices, as without prioritization of needs, standardized workflows and platform-level oversight, the potential benefits of microservices might not be fully realized.

Despite the challenges for researchers to access and study industry-level systems, there is a distinct need for more peer-reviewed literature on the effects of adopting microservices in full-scale software engineering [10]. Further research can unlock more insight into the long-term effects of microservices by developing standardized metrics for evaluating the effectiveness of a microservices implementation in reaching its initial goals. Investigating long-term maintenance strategies for evolving microservices systems, and studying the impact on teams and organizational culture can help bridge the gap between the promises and the realized outcomes of microservices.

7. REFERENCES

[1] Iso/iec 25010. `https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?limit=5&start=7#:~:text=Adaptability%20%2D%20Degree%20to%20which%20a,its%20capacity%20to%20handle%20variability.` "Retrieved November 23, 2024".

[2] G. Blinowski, A. Ojdowska, and A. Przybyłek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10:20357–20374, 2022.

[3] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. Microservices in industry: Insights into technologies, characteristics, and software quality. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 187–195, 2019.

[4] M. E. Conway. How do committees invent? *Datamation*, April 1968.

[5] 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.

[6] M. Fowler. Microservice premium. https://martinfowler.com/bliki/MicroservicePremium.html, 05 2015.

[7] J. Ghofrani and D. Lübke. Challenges of microservices architecture: A survey on the state of the practice. 05 2018.

[8] J. Gray. A Conversation with Werner Vogels. https://queue.acm.org/detail.cfm?id=1142065, 2006. Retrieved November 13, 2024.

[9] W. Hasselbring and G. Steinacker. Microservice architectures for scalability, agility and reliability in e-commerce. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 243–246, 2017.

[10] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[11] V. Lenarduzzi and O. Sievi-Korte. On the negative impact of team independence in microservices software development. In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, XP '18, New York, NY, USA, 2018. Association for Computing Machinery.

[12] J. Lewis and M. Fowler. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html/, 2014. Retrieved November 11, 2024.

[13] S. Newman. *Building Microservices*. O'Reilly Media, Inc., 1st edition, 2015.

[14] J. Nicholas. Death Star Architecture — mrtortoise.github.io. https://mrtortoise.github.io/architecture/lean/design/patterns/ddd/2018/03/18/deathstar-architecture.html, 2018. Retrieved November 14, 2024.

[15] R. Osowski. Introduction to microservices. https://developer.ibm.com/tutorials/cl-ibm-cloud-microservices-in-action-part-1-trs/, 07 2024.

[16] M. Paasivaara, C. Lassenius, and V. T. Heikkilä. Inter-team coordination in large-scale globally distributed scrum: Do scrum-of-scrums really work? In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 235–238, 2012.

[17] C. Pahl and P. Jamshidi. Microservices: A systematic mapping study. pages 137 – 146, Setúbal, Portugal, 2016. SciTePress.

[18] M. R. Saleh Sedghpour, C. Klein, and J. Tordsson. An empirical study of service mesh traffic management policies for microservices. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ICPE '22, page 17–27, New York, NY, USA, 2022. Association for Computing Machinery.

[19] E. Sörensen. Language diversity in microservices: a case study at skatteverket. https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1565803&dswid=490, 2021. "Retrieved November 23, 2024".

[20] D. Taibi and V. Lenarduzzi. On the definition of microservice bad smells. *IEEE Software*, 35(3):56–62, 2018.

[21] D. Taibi, V. Lenarduzzi, and C. Pahl. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, 4(5):22–32, 2017.

[22] H. Vural, M. Koyuncu, and S. Guney. A systematic literature review on microservices. pages 203–217, 07 2017.

[23] Z. Xiao, I. Wijegunaratne, and X. Qiang. Reflections on soa and microservices. In *2016 4th International Conference on Enterprise Systems (ES)*, pages 60–67, 2016.