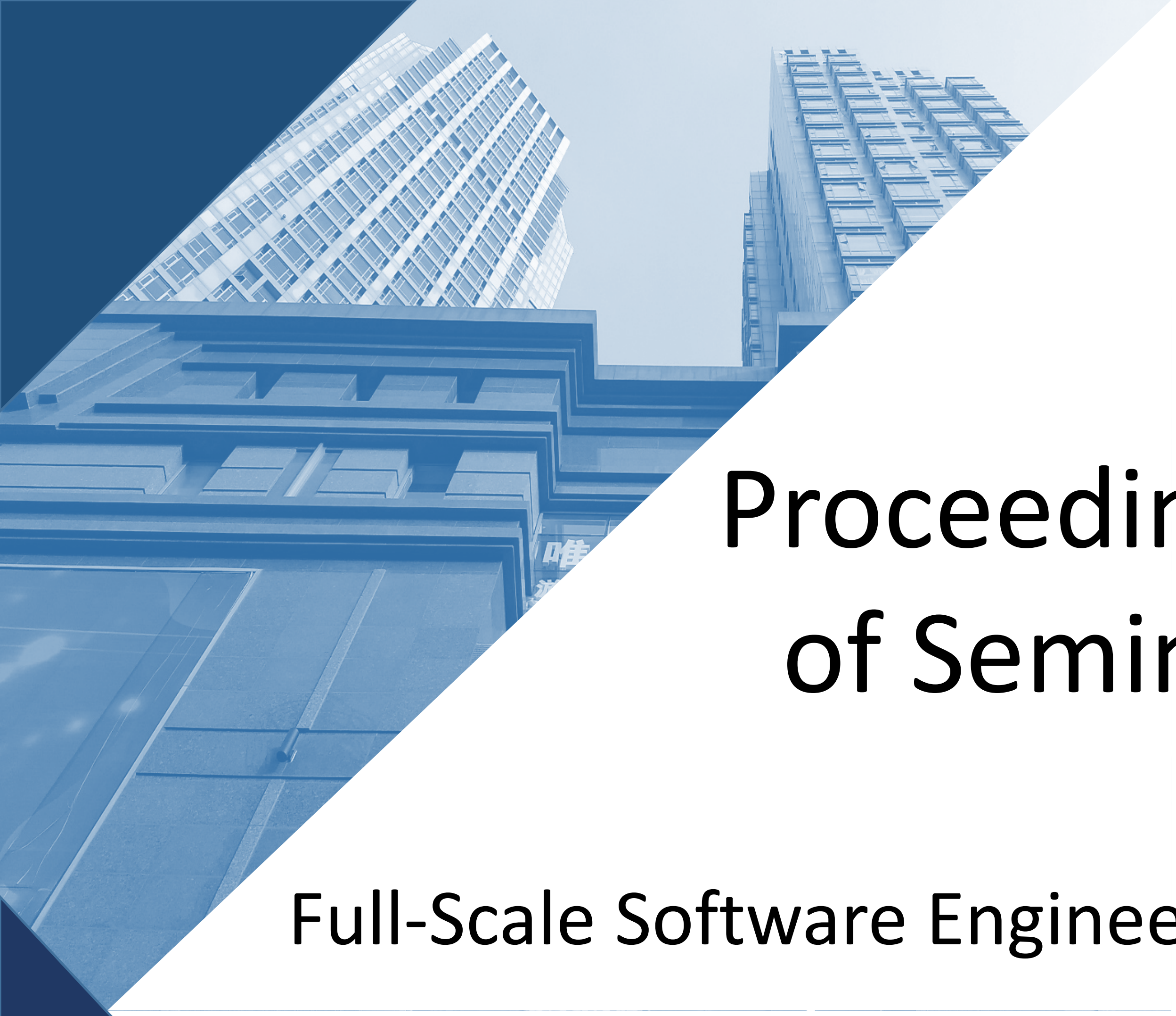

Proceedings of Seminar

Full-Scale Software Engineering

2023

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

Table of Contents

The relation between Modernization and EA Debt

Bashmund Shah
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
bashmund.shah@rwth-aachen.de

Utkarsh Dubey
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
utkarsh.dubey@rwth-aachen.de

ABSTRACT

The Software development process is dynamic and may include perpetual evolution of legacy systems which are prone to logical and technical fallacies. The technological evolution of such legacy systems includes Modernization i.e. restructuring and modifying the system. Modernization aims to overcome Technical Debt which is the cost incurred when expedient solutions are implemented. Even though Modernization affects the entire enterprise, the impact of Modernization on business has not been studied as Technical Debt alone does not encompass business implications. Technical Debt is a subset of Enterprise Architecture Debt and Enterprise Architecture Debt additionally focuses on business and data. Since there is significant research on the causes, minimization and impact of Modernization on Technical Debt, we want to use the relation of Technical Debt and Enterprise Architecture Debt to relate Modernization with Enterprise Architecture Debt. This research is focused on deducing the impact of Modernization on the entire enterprise and the relation between Modernization and Enterprise Architecture Debt. To do that, we performed a literature review of 21 papers to support our findings.

Categories and Subject Descriptors

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

Keywords

Modernization, Technical Debt, Enterprise Architecture Debt

1. INTRODUCTION

In today's world, the ever-changing requirements have led to the iterative development of software. This involves continuous software development to adapt to changing requirements. It is undeniable that the software development process is dynamic, a perpetual evolution of software. Such development cycles are prone to logical and technical fallacies which could lead to rushed or unplanned development.

This leads to the introduction of **Technical Debt**. Technical Debt is the cost that is incurred when expedient solutions are implemented. These solutions might seem practical at the time but hidden beneath is a liability that has to be taken care of. Initially, it might seem like a minor issue that can be dealt with at a later date, however, this negligence leads to the accumulation of debt. It affects the quality, stability and functionality of the system. It can be present at numerous levels such as in the code, data structure, or data itself. With time, there is a need to take care of such issues. This can be done in numerous ways which may involve refactoring, discarding the software, rewriting the code, or using a different stack, tools and technologies. There is no single defined way of dealing with the issue. Moreover, the process itself is referred to with different terms such as migration, modernization, evolution, re-structuring, or re-engineering etc. These terms may vary slightly in their definition provided by various literature, but all of them have common goals [5][10].

Modernization is the process of updating or upgrading a software system that cannot keep up with today's standards in terms of performance, goals, or tech stack. This mainly involves systems known as legacy systems that have been developed in the past and have accumulated Technical Debt over time. Not only such systems have deteriorated in quality and performance, but they do not meet the enterprise goals anymore as they lack the features to keep up with today's competition [6][19][12].

Enterprise Architecture Debt refers to the accumulated challenges, inefficiencies, and shortcomings within the overall Enterprise Architecture of an organization, resulting from deferred decisions, outdated technologies, and sub-optimal design choices. It not only involves the technical aspect of the software system but also highlights the big picture in terms of hindrances to the architecture, enterprise, business goals and ideals [4][10].

Now the question arises "Given a relation between Modernization and EA Debt, is the relation to Technical Debt different?". This research mainly aims to answer this question. The goal is to identify similarities and differences between Modernization and Enterprise Architecture Debt. Furthermore, the research is aimed to determine the impact of Modernization on Enterprise Architecture and its relation with EA Debt.

2. RELATED WORK

2.1 Modernization

System evolution covers a range of development activities from adding additional fields in a database to completely re-implementing a system. These activities can be divided into three categories: maintenance, modernization, and replacement [20]. Regarding software evolution, Lehman's first law [14] states that software has to be adopted otherwise it would not give satisfactory results. In these cases, Modernization helps to keep things up-to-date and in use. Modernization involves more extensive changes than maintaining the software itself. It includes restructuring the system, enhancing functionality, or modifying software.

According to [12, 15], there are three main reasons for modernizing the systems:

- **Addressing deterioration:** To counteract the deterioration of the internal structures of the system and provide a large number of entry points for integration with other systems.
- **Challenges in legacy systems:** Legacy systems pose difficulties in compatibility, security, enhancement, and complexity, making their maintenance challenging.
- **Supplier dependence:** Dependence on the supplier becomes a challenge when technologies become obsolete, making it difficult to hire qualified labour.

Overall, Modernization is used when a legacy system requires more pervasive changes than those possible during maintenance, but it still has business value that must be preserved. It is crucial to meet the current business needs of the organization.[19] Most of the previous works on Modernization have addressed the need for Modernization and how to perform it. However, there is a significant gap in studies investigating the relationship between Modernization and Enterprise Architecture.

2.2 Technical Debt

The term *Technical Debt* has various interpretations. However, the most common version is the one given by Cunningham.

Cunningham defined Technical Debt as follows [3]:

> "Technical Debt includes those internal things that you choose not to do now, but which will impede future development if left undone. This includes deferred refactoring."

The definition of Technical Debt was further extended by Seaman et al. [9] to show other kinds of debts in software development, such as test debt, people debt, architectural debt, requirement debt, documentation debt, etc.

According to [5], Technical Debt is used as a uniform communication tool that allows us to measure and keep track of debt, which eventually should help find a suitable solution to upcoming challenges. In this case, it should also reflect different viewpoints, including the stakeholder's perspective, to allow effective collaboration.

Technical Debt has shown its benefits in estimating deficits of software construction and providing a tool for decision-making [9]. However, the metrics used for Technical Debt must be generalized to be applicable and helpful for the entire enterprise and every scenario. This encompasses the different layers of Enterprise Architecture (EA), including business and IT with its systems and strategies in particular. Therefore, it becomes possible to estimate the consequences of EA implementation failures using a new term, EA Debts [10].

2.3 Enterprise Architecture

There are many divergences between Enterprise Architecture (EA) definitions, and the nature of some of them is significant [18]. The term "enterprise" in EA is interpreted by some as synonymous with "enterprise systems," while others perceive it as comparable to "business" or "organization." The understanding of the term "architecture" has even less uniformity. The prevailing interpretation of the term "architecture" in EA is a collection of artifacts (models, descriptions, etc.) that establish the guidelines of how the enterprise should function or provide an as-is model of the enterprise [13].

These divergences in Enterprise Architecture definitions have several consequences, like the confusion in the role of an enterprise architect and the lack of common understanding of EA. However, EA helps to face "ongoing and disruptive change" by attempting to align IT and business strategy [18].

Overall, it can be concluded that there are different definitions of EA in various literature, but it is more often described for the enterprise orientation. Hence, there is a common understanding of EA and a basis for communication and discussion.

2.4 EA Layers

According to Simon Hacks et al., [10], since there are multiple definitions of Enterprise Architecture (EA), EA should be regarded as a set of artifacts that are aggregated. The artifacts themselves and their importance for a specific enterprise may differ, so the focus should be on the aspects in particular, which are general enough to be applied to the majority of EAs.

This is done by mapping the TOGAF Standard version 9.2 [17] with the different layers of Enterprise Architecture suggested by Winter and Fischer [21]

Following are the EA layers which are general enough to be applied to most organizations:

- Business Layer
- Data Layer
- Application Layer
- Technical Layer

Considering EA as a set of artifacts, it can contain more or fewer than the mentioned artifacts, but it is easy to add and remove artifacts later [10].

2.5 EA Debts

The term "EA Debt" came about by taking into account the combination of "Enterprise Architecture" and the "Technical Debt". Winter et al.[21] pointed out that "Most of the artifacts in EA can be represented as aggregation hierarchies." Combining this with the definition of "Technical Debt" given by Cunningham [3], it can be concluded that

EA Debt is obtained by taking the aggregation of debt obtained from each layer of an enterprise. However, Simon et al. [10] argued that adding up each artifact would not give a concrete overview of the current situation, and one must weigh every part of every artifact to estimate the EA Debt. Also, there is no uniform weighting function because it depends on the concrete EA, the organization, and the dependencies between the artifacts.

Therefore, the following is the definition of EA Debt given by Simon et al. [10]:

> "Enterprise Architecture Debt is a metric that depicts the deviation of the currently present state of an enterprise from a hypothetical ideal state."

Technical Debt can be related to EA Debt as a sub-domain of it. However, there are also issues related to EA Debts on software and technology aspects that are still not covered by Technical Debt [12]. Overall, based on the above discussion it can be implied that when the artifact or an element of the artifact is not implemented properly for the optimal ideal situation then EA Debt arises [10].

3. METHODOLOGY

3.1 Scientific Literature Review

To answer the desired research questions, a scientific literature review method is chosen for this purpose. The focus was mainly on the following terminologies: Modernization, Enterprise Architecture Debt, and Technical Debt. The following keywords were used in several different combinations to search for papers:

- "Modernization" OR "Enterprise Architecture" OR "Technical Debt"
- "Modernization" AND "Enterprise Architecture" AND "Technical Debt"
- "Modernization" AND "Enterprise Architecture"
- "Modernization" AND "Enterprise Architecture"
- "Modernization" AND "Enterprise Architecture"
- "Enterprise Architecture Debt" AND "Technical Deb"

It is worth noting that both the AND and OR operators were employed for specific reasons. The AND operator was chosen due to its tendency to produce fewer results, focusing strictly on the relationship between the specified terms. However, the inclusion of the OR operator was deemed necessary as it facilitates the analysis of each term individually.

DB#	DB Name	No. of Papers
1	IEEE Explore	4270
2	Scopus	5425
3	Dblp	927
4	ACM (Digital Library)	897
5	IET (Digital Library)	765
6	Science Direct	694

Table 1: List of databases and their corresponding number of papers

Modernization, Technical Debt, and Enterprise Architecture Debt. With current search criteria, the following results were gathered as shown in Table 1.

Since the results are too big, different filtration techniques must be applied.

3.1.1 Filtration Techniques

To filter out the results, it is important to objectively analyze all search results based on the following criteria:

1. Relevance to the research
2. Language of the paper
3. Relevant publication topics
4. Subject type
5. Document type
6. Redundant or duplicate results
7. Manual Filtration

The above-mentioned criteria were applied by per- forming the following steps in sequential order:

1. **Relevance to the research:** Throughout the research, the goal was to ensure that the papers were relevant to the research question. This involved choosing the right search terms and making use of other techniques which are mentioned below and prioritizing the relevance of the filtered papers
2. **Database Filtration:** the following criterion were part of database filtration by making use of provided filtration techniques:
   (a) **Language of the paper:** For the sake of simplicity, papers only written in **English** language were selected.
   (b) **Subject type:** This criteria is relatively self-explanatory. The primary interest was mainly in the field of Computer Science. Interestingly enough, several other subject types made use of the term: technical debt. On the other hand, modernization is a versatile term that can be used in several ways in different contexts. Therefore, it was very important to make sure that these papers revolved around Computer Science.
   (c) **Relevant publication topics:** Some databases had the functionality to filter the results based on publication topics. These databases were IEEE Explore and Scopus. The topics chosen for filtration were software maintenance, software quality, project management, software architecture, software development management, systematic literature review, refactoring, and software engineering.
   (d) **Document type:** The focus was mainly on Conference papers, articles and journals. The reason behind this criterion is that aforementioned document types are published more frequently unlike books. Therefore, they provide more up-to-date information compared to books. Furthermore, these documents are specialized and have a deeper focus on specific research questions.

3. **Redundant or duplicate results:** As the search was carried out on multiple databases using the same filtration techniques, it is not unlikely search results end up with a redundant set of papers from different databases. Therefore, it had to be ensured that all the documents from every database were unique and that no such documents were repeated.

4. **Manual Filtration:** This process required going through the search results manually to identify which papers were more suited for the research. The goal was to find documents that provided condensed information and covered a wide range of terms relevant to the research. Also, it was ensured that the terms are used in some context which is related to the research questions of our paper.

   This was done using the following evaluation techniques:

   (a) **Review the title:** For each paper, the title was reviewed to get a rough idea of the context.

   (b) **Review the abstract of the paper:** For each paper, the abstract was reviewed to understand the relevance of the paper to the research objective.

   (c) **Searching the relevant keywords in the paper:** For each paper, relevant keywords were searched to understand the context of their usage throughout the paper.

   (d) **Review the conclusion:** For each paper, the conclusion was reviewed, and a decision was made whether to include/reject the paper.

   (e) **Bias reduction:** To reduce bias, the databases were pruned individually. In case of difference of opinions, it was thoroughly discussed within the authors why a specific document was included or excluded from the list.

After going through an extensive scientific literature review, the following numbers of papers were shortlisted from each database, as shown in the Table 2.

DB#	DB Name	No. of Papers
1	IEEE Explore	7
2	Scopus	2
3	Dblp	3
4	ACM (Digital Library)	5
5	IET (Digital Library)	3
6	Science Direct	1

Table 2: Filtered list of papers

Figure 1 presents a comprehensive depiction of the search and filtration procedures. Initially, the search yielded approximately 13,000 results. However, following the prescribed steps, the final count was refined to 2413 papers. Despite this, the applied filtration steps did not ensure 100% relevant papers. Subsequently, manual filtration became imperative to procure research materials that could meet qualitative standards within the specified content constraints of the seminar. After a meticulous evaluation of various factors outlined in the fourth step, namely manual filtration, the final number was further reduced to 21 results.

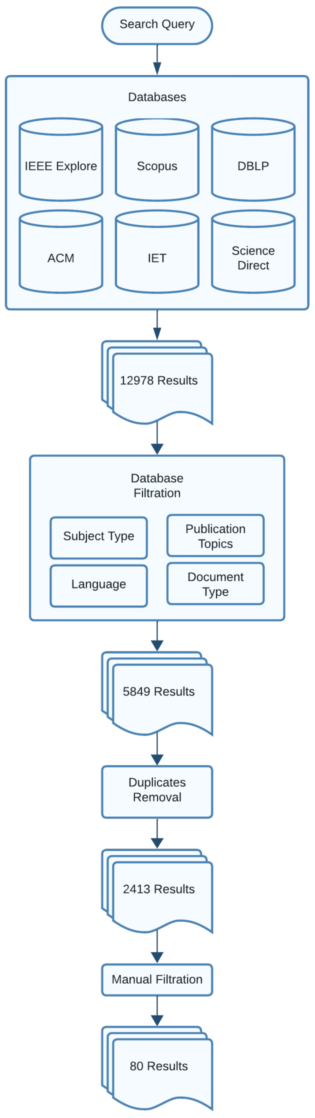

Figure 1: Overview of Search Results Filtering

4. MODERNIZATION AND ENTERPRISE

Large enterprises, grown over decades have difficulties in ensuring business innovation and cost-efficiency at the same time. They often have outdated and difficult-to-maintain complex systems with overlapping functionality. Therefore, these enterprises initiate strategic Modernization programs to modernize essential parts of their enterprise. Such programs aim to replace outdated, complex systems or software with loosely coupled services, re-used for business processes across all relevant regions of the world [8].

Table 1 shows the impact of Modernization programs on an enterprise. We can conclude from the figure that the Modernization of an enterprise provides the following benefits:

- Cost-reduction
- Business process optimization
- Business agility and faster time-to-market

In this study, the impact of Modernization programs on an enterprise is used to assess the impact of Modernization on the individual layers of an enterprise.

AS-IS	**TARGET**
Slow reaction to business needs	high business agility
”IT Silos” with redundancies	Reduced-redundancies through common platforms
End-of-life legacy application (20+ years)	Moving away from legacy applications
High Development, Testing and Deployment costs	Cost Efficient Development and operations

Table 3: Impact of modernization programs on an enterprise

4.1 Impact of Modernization on the Business Layer of an Enterprise

The business layer of an enterprise includes the business processes and functions of an enterprise. These processes and functions are responsible for ensuring that the enterprise provides the desired business outcome [7]. According to [19], Modernization involves more extensive changes than maintaining the software like re-structuring and modifying the system. Whenever any system or software is restructured or there are changes in the functions that are used to develop the software, there would be changes in the business processes that are used by the enterprise. Therefore, it can be inferred that Modernization could reshape the business processes and functions of an enterprise which consequently makes the business architecture more agile and flexible. The use of legacy systems by an enterprise provides epochal business value, which is not suitable when the business needs and requirements are expected to be agile. Apart from decreasing the efficiency of the business, legacy systems also hold back the enterprise from creating the best possible business architecture.

Suman et. al [11] point out that during Modernization, the existing application is recreated with further enhancements and heightened technical capabilities in the cloud, which in turn simplifies the business's streamlined processes. This helps the enterprise to deliver more consistent performance and improved functionalities. Therefore, if Modernization tasks are performed appropriately then the system would provide more consistent and improved business outcomes.

Overall, we can conclude that Modernization improves the agility and flexibility of the enterprise's business layer. Also, if the Modernization tasks are performed appropriately by the enterprise or the organization, then it can also improve the business outcome.

4.2 Impact of Modernization on the Data Layer of an Enterprise

The data layer of an enterprise is responsible for providing secure information exchange and sharing between stakeholders and access to all information to anyone who is authorized to see it [7]. Now, in the modern day, as data is becoming more central to businesses, it is clear that outdated data architectures can no longer keep up with the demands of the modern enterprise. Kandukuri et. al [12] point out that enterprises turn to Modernization when their data infrastructure becomes too inflexible for today's business demands. Modernization involves various tasks to overcome the data exchange problems of legacy systems and improve data quality. This involves updating an organization's data management infrastructure, tools, and processes to keep pace with evolving technologies and best practices. These strategies help enterprises improve legacy data quality while applying a first-principles approach to manage data moving forward.

Higher data quality through Modernization activities would eventually improve decision-making throughout the enterprise by delivering more timely and accurate insights to everyone in the organization. Also, Modernization offers near-term cost savings in the data architecture of an enterprise. For instance, companies can significantly reduce data costs by migrating data to cloud platforms and reducing the reliance on data warehouses.

Overall, it can be inferred that Modernizing an enterprise can significantly improve the data exchange mechanism of the enterprise and also keep the data architecture up to date with the latest standards.

4.3 Impact of Modernization on the Application Layer of an Enterprise

The application layer of an enterprise focuses on providing services-oriented solutions for the application software that meet the business needs. These solutions are chosen and implemented based on the optimal combination of complexity, business value and costs. The application layer of an enterprise generally includes all the IT components that the stakeholders directly interact with in the automation of the business process [7].

Legacy systems may have a cumbersome user interface which is mainly because the solutions provided by these systems rely on outdated technologies that have poor scalability and limited adaptability. Moreover, they provide inefficient performance that hinders productivity[19]. Therefore, legacy systems are a major blocker to the goal of optimal user experience for an enterprise.

The Modernization of the application layer of an enterprise is motivated by the need to change or redesign the application to improve the user experience. The activities to modernize the application involve improving or replacing legacy applications with more efficient and scalable solutions. For instance, efficient solutions using modern development frameworks or micro-services as an architectural approach for developing software applications as a collection of small, independent services that communicate with each other over a network instead of building a monolithic application where all the functionality is tightly integrated into a single code base [12]. These efficient solutions are often approached as part of a broader digital transformation initiative. These initiatives are a process of using digital technologies to create new business processes and customer experiences (or modify existing ones) to meet changing business and market requirements. It often involves the integration of digital technologies and customer-centric approaches to improve business operations and competitiveness [15].

Hence, it can be inferred that Modernization improves the application layer by providing more efficient and scalable solutions which aim to improve the overall user experience.

4.4 Impact of Modernization on the Technology or (Technical) layer of an Enterprise

The technology layer (also known as the technical layer or infrastructure layer [21]) is responsible for the definition and design of the current and future technology infrastructure of the described enterprise, including hardware, software and application platform [7]. The technology layer also supports the application layer of the enterprise to ensure that the system satisfies a specified set of requirements.

Legacy systems tend to struggle to adopt and adapt to emerging technologies. This is mainly because legacy systems rely on outdated technologies that are built using older tools and systems which are incompatible with modern standards and best practices [12]. Therefore, legacy systems may not easily adapt to changing business requirements, making them ill-suited for modern, dynamic organizations.

In such cases, Modernization activities allow enterprises to move away from legacy systems and utilize emerging technologies. The use of emerging technologies improves the productivity of the enterprise developing the software and ensures compatibility with modern standards and best practices. For instance, the use of technologies like Docker Containers and Kubernetes delivers speed, simplicity and self-service for the developers of the enterprise. The adoption of such technologies by an enterprise also motivates its employees to learn new skills and work towards their development which in turn leads to the growth of the enterprise [19].

Overall, it can be inferred that Modernization improves the technology layer by adopting new emerging technologies to improve the productivity of the enterprise developing the software and supporting the application layer of the enterprise to ensure that the system satisfies a specified set of requirements.

5. EA DEBT FACTORS AND MODERNIZATION IMPLICATIONS

One of the biggest goals in the software development process for an enterprise is to ensure that the organization works to achieve its business goals. One such obstacle is the effect of development liabilities' build-up that might seem initially insignificant. However, the continuous growth of such incurring debt eventually proves to be one of the most significant hurdles for an enterprise. This term is known as **Enterprise Architecture Debt**.

Furthermore, EA Debts affect enterprises in numerous ways that require major reforms to address this issue. Modernization serves as a solution to this problem. However, it must be ensured correct measures are taken, otherwise, it could have adverse effects. The factors mentioned below show that the relationship between Modernization and EA Debt is intertwined.

5.1 Business Factors

5.1.1 Uncertainty of use cases in the beginning

At the beginning of software development, there is no proper infrastructure for development and the business idea is yet to be nourished. This period is most vulnerable to quick changes which lead to rapid development, albeit, unorganized. This contributes to the build-up of Enterprise Architecture Debt [16][19].

This uncertainty needs strategic Modernization techniques. There are numerous ways that such reforms can be done. However, each strategy depends on the given circumstances. It is likely that once the business is matured, it could be revamped to better cater for the needs of current use cases.

5.1.2 Business Evolution

There are numerous scenarios where business evolves into something completely new. It could be a merger situation or a major client's feature request that would require accelerated development into a new direction. This sudden shift is unanticipated and unstructured. This affects the integrity of the system where the focus is to provide the new feature conforming to new business goals instead of the stability of the overall system [16][19].

In such cases, it is very important to make the system adaptable to the enterprise changes. The build-up of EA Debt could be significant if it is not handled timely. Oftentimes, it has been seen that every new product launch is followed by re-factoring at the code and data layer level. This helps in making the system stable and flexible for future changes.

5.1.3 Time Pressure

There is a very fine line between high time pressure for improved efficiency and reduced quality. One has to balance the applied pressure to ensure high-quality development at all times with more efficient methods. However, under numerous circumstances, it has been seen that product managers have to give strict deadlines to meet the goals.

Rushed development comes at the cost of allocating additional time for refactoring. It is important to realize that unrealistic deadlines do not improve development's pace. Modernization also involves changing development practices that ensure the stability and quality of the software. Therefore, it can be concluded that such reforms can also be applied at the organizational level.

5.1.4 Priority of features over the product

To attract more customers, it has been a common tactic to incorporate as many features into the product as possible. This clearly emphasizes quantity over quality. However, in the long run, the lack of quality leads to incurring EA Debt which would require major reforms.

Modernization efforts should realign the enterprise's priorities. Emphasizing product stability as a key priority is essential for long-term sustainability. Modernization strategies should include a reassessment of feature development practices, ensuring that each addition aligns with a stable and well-architected foundation.

5.2 Design & Architecture Documentation

Every software development cycle involves defining software specifications and requirements. The lack of proper or inarticulate documentation leads to misinterpretation by developers. In most cases, such mistakes are discovered later on and the cost of rectifying such mistakes is much higher than early stages.

As Modernization covers the entire enterprise, it also includes assigning correct methodologies for effective documentation. It should emphasize the importance of comprehensive documentation from the beginning. Otherwise, this would become a liability at a later stage. Furthermore, such Modernization initiatives prevent misinterpretations that could lead to potential EA Debt.

5.3 Re-use of legacy systems

Using a legacy system may seem to be a good solution for a quick fix. However, in the long term, it could be seen that such systems differ in architecture and deviate from the programming standards of the in-house system based on modern frameworks. It becomes complex to keep such micro-services to be compatible [16][20].

This corresponds with an important term known as ”Wrapping” which is a widely used Modernization strategy which involves using a legacy system under the disguise of a wrapper. It is often referred to as black-box Modernization which requires zero to very little knowledge of the legacy system.

5.4 Parallel Development

Multiple development teams working in parallel on the same feature leads to contradictory implementation methods, inefficient synchronization techniques, and varied approaches. This branches into two different directions where making amendments is costly and time-consuming.

Such scenarios can be reduced with proper planning. It would require changing current development practices. However, it is not the only solution. Such issues could be resolved with proper refactoring but it could be more time-consuming. Each solution has its advantages and disadvantages and it depends a lot on given circumstances to choose the right solution.

5.5 In-complete Refactoring

It has been a common practice where feature requests are prioritized over current refactoring tasks. The refactoring tasks were created in the first place to reduce the build-up of incurring Enterprise Architecture Debt but leaving such task halfway through leads to further build-up of EA Debt [16][1].

Refactoring constitutes a crucial component of Modernization strategies. Incomplete refactoring not only fails to address existing Technical Debt but also contributes to its further accumulation, ultimately leading to more Enterprise Architecture Debt.

5.6 Technology Evolution

With ever-changing technological changes where there is innovation in every aspect of technology, it is hard to build a stable product. It is necessary to adapt to technology evolution otherwise the product likely feels less modern to current standards. Most enterprises fear being left behind by their competitors, this fear leads to abrupt and rushed development strategies to accommodate technological evolution which leads to incurring EA Debt [2][20].

Modernization plays an important role in mitigating the impact of technology evolution on Enterprise Architecture Debt. Adopting an iterative and proactive approach to development strategies to keep up with technological evolution is one step toward Modernization.

5.7 Human Factor

Last, but not least, software engineering is susceptible to mistakes due to inexperience, ignorance, or lack of expertise in the relevant field. This leads to an unstable product.

To address such problems, periodic re-factoring is one such solution that can mitigate the problem. However, if the problem remains unaddressed, it would require re-engineering the whole product. In the end, it depends on the magnitude of the problem and ensuring suitable Modernization measures are taken place.

6. CONCLUSION

The comprehensive examination of the scientific literature involved the evaluation of around 21 papers, aiding in the identification of gaps in the field of EA Debt and Modernization. Through the analysis of these papers, connections were established to address the two research questions.

Firstly, Modernization impacts every layer of an enterprise in some or the other way. The extent of the impact on a specific layer of an enterprise depends mainly on the Modernization method that is being implemented. For instance, Modernization activities like re-structuring or modifications in the functions used to develop a system affect the business layer. Similarly, upgrading the organization's data architecture affects the data layer, replacing legacy systems with more efficient solutions affects the application layer and the use of emerging technologies affects the Technical layer of an enterprise.

Finally, EA Debt and Modernization are very deeply related. They are intertwined in the process of software development. It could be stated that in some cases, they act as the cause and effect of one another. For Example - Modernization strategies include refactoring that helps to minimize the EA Debt. Similarly, Modernization also plays a key role in reducing the impact of technology evolution on EA Debts.

Overall, Modernization impacts every layer of an enterprise and can minimize the Enterprise Architecture Debt.

7. FUTURE PROSPECTS

Moving forward, there are several potential directions for future research:

7.1 Developing Metrics for EA Debt

It can be deduced from the given research that there is a need for a uniform weighting function to estimate EA Debt. Future research could focus on developing such metrics that can be generalized to be applicable and helpful for the entire enterprise and every scenario.

7.2 Exploring the Role of Human Factors

The paper has pointed out that software engineering is susceptible to mistakes due to inexperience, ignorance, or lack of expertise. Future research could explore this issue in more depth, investigating how these human factors can be mitigated to produce a more stable product.

7.3 Investigating the Impact of Modernization on Business Outcomes

As it has been discussed if Modernization tasks are performed appropriately, the system would provide more consistent and improved business outcomes. Future research could investigate this claim further, exploring how different modernization strategies impact business outcomes.

8. REFERENCES

[1] 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.

[2] H. Cervantes and R. Kazman. Software archinaut: a tool to understand architecture, identify technical debt hotspots and manage evolution. In *Proceedings of the 3rd International Conference on Technical Debt*, pages 115–119, 2020.

[3] W. Cunningham. The wycash portfolio management system. *ACM Sigplan Oops Messenger*, 4(2):29–30, 1992.

[4] S. Daoudi, M. Larsson, S. Hacks, and J. Jung. Discovering and assessing enterprise architecture debts. *Complex Systems Informatics and Modeling Quarterly*, (35):1–29, 2023.

[5] 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*, pages 50–60, 2015.

[6] T. C. Fanelli, S. C. Simons, and S. Banerjee. A systematic framework for modernizing legacy application systems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 678–682. IEEE, 2016.

[7] B. Firmansyah and A. A. Arman. A systematic literature review of regtech: Technologies, characteristics, and architectures. In *2022 International Conference on Information Technology Systems and Innovation (ICITSI)*, pages 310–315. IEEE, 2022.

[8] F. J. Frey, C. Hentrich, and U. Zdun. Capability-based service identification in service-oriented legacy modernization. In *Proceedings of the 18th European Conference on Pattern Languages of Program*, pages 1–12, 2013.

[9] Y. Guo. *Measuring and monitoring technical debt.* University of Maryland, Baltimore County, 2016.

[10] S. Hacks, H. Höfert, J. Salentin, Y. C. Yeong, and H. Lichter. Towards the definition of enterprise architecture debts. In *2019 IEEE 23rd International Enterprise Distributed Object Computing Workshop (EDOCW)*, pages 9–16. IEEE, 2019.

[11] S. Jain and I. Chana. Modernization of legacy systems: A generalised roadmap. In *Proceedings of the Sixth International Conference on Computer and Communication Technology 2015*, pages 62–67, 2015.

[12] P. Kandukuri. Software modernization through model transformations. In *First International Conference on Artificial Intelligence and Cognitive Computing: AICC 2018*, pages 165–174. Springer, 2019.

[13] L. Kappelman, T. McGinnis, A. Pettite, and A. Sidorova. Enterprise architecture: Charting the territory for academic research. 2008.

[14] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution-the nineties view. In *Proceedings Fourth International Software Metrics Symposium*, pages 20–32. IEEE, 1997.

[15] P. L. Leon and F. E. A. Horita. On the modernization of systems for supporting digital transformation: A research agenda. In *XVII Brazilian Symposium on Information Systems*, pages 1–8, 2021.

[16] A. Martini, J. Bosch, and M. Chaudron. Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study. *Information and Software Technology*, 67:237–253, 2015.

[17] G. F. Nama, D. Kurniawan, et al. An enterprise architecture planning for higher education using the open group architecture framework (togaf): Case study university of lampung. In *2017 Second International Conference on Informatics and Computing (ICIC)*, pages 1–6. IEEE, 2017.

[18] P. Saint-Louis, M. C. Morency, and J. Lapalme. Defining enterprise architecture: A systematic literature review. In *2017 IEEE 21st international enterprise distributed object computing workshop (EDOCW)*, pages 41–49. IEEE, 2017.

[19] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing legacy systems: software technologies, engineering processes, and business practices.* Addison-Wesley Professional, 2003.

[20] N. H. Weiderman, J. K. Bergey, D. B. Smith, and S. R. Tilley. Approaches to legacy system evolution. *Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa*, 1997.

[21] R. Winter and R. Fischer. Essential layers, artifacts, and dependencies of enterprise architecture. In *2006 10th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW'06)*, pages 30–30. IEEE, 2006.

Overview and Comparison of Current Automated Threat Modeling Approaches

Erik Wrede
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
erik.wrede@rwth-aachen.de

Szymon Habrainski
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
szymon.habrainski@rwth-aachen.de

ABSTRACT

The advancement of threat modeling through automation presents a transformative opportunity for enhancing software security. Current literature extensively explores various threat modeling processes and their applications across diverse environments. However, a focused review of automating threat modeling—a crucial step for improving cost-effectiveness and accessibility—has not been conducted yet. This paper presents a literature review of cutting-edge approaches to threat modeling automation. We categorize automation techniques into three distinct areas: the creation of dynamic threat catalogs, ontology-based threat modeling strategies and the use of configuration assets for automated threat identification. Our discussion evaluates these methods based on their readiness for industry use, weighing the necessary effort against their potential for broad application across different software development projects. The automation provided by the approaches was found to be highly beneficial for adopting threat modeling. However, we recognize that further research is necessary to integrate these approaches into effective industry processes. We suggest combining multiple methods to create a fully automated threat modeling process, underscoring its importance in advancing software security.

Categories and Subject Descriptors

[**Security and privacy**]: Software security engineering

Keywords

Threat Modeling, automation, artificial intelligence

1. INTRODUCTION

The increasing digitization of the modern world implies unique security risks to the software infrastructure powering it. Software security threat modeling is a framework aimed at systematically identifying potential threats and security risks to an application during the planning and development phase of the software development lifecycle [8, 22]. Threat modeling (TM) is typically applied in a three-step process, encompassing the modeling of the system architecture, threat identification and prioritization, and the formulation of mitigation techniques [7]. Process execution is commonly supported by frameworks such as STRIDE [19] or MITRE ATT&CK [17].

While threat modeling is becoming a common tool in software development projects [26], its practical adoption in the industry still faces significant challenges. Yskout et al. describe the common practice as relying on "ad-hoc" and pragmatic "whiteboard hacking", dependent on the participants of the meeting instead of a formalized, scientifically founded process [27]. The application of a formalized Threat Modeling Process (TMP) has been identified as costly, time-consuming, and heavily reliant on expert knowledge [22, 21]. Additionally, artifacts of the process, such as diagrams or threat catalogs, are often incomplete or unclear, rendering threat modeling incapable of providing an overview of all system threats [26, 10]. Particularly, the cost and time factors represent significant hindrances to adopting threat modeling. With the rise of agile software development techniques, these challenges are exacerbated due to the fast-paced, iterative nature of agile software development lifecycles (SDLCs). This results in additional overhead in re-modeling the system for the next threat modeling iteration, leading to repetitive meetings [22].

The automation of the identification and modeling phases of the TMP may help address these challenges by reducing the need for security expert involvement and increasing the efficiency and productivity of the process [11]. Automated Threat Modeling (ATM) can facilitate faster remodeling of infrastructure changes and reduce human error during the process [8].

Given the absence of a dedicated literature review specifically focusing on automation in threat modeling, this paper performs an unstructured literature review of existing approaches for ATM, compares them, and discusses their suitability for industry use while also outlining potential future work in this area. Our review is centered around the following two research questions aimed at accurately capturing the forefront of advancements in ATM:

- **RQ1**: What is the current state of research in ATM?
- **RQ2**: How do current methods differ regarding their applicability in industry and threat reporting accuracy?

The remainder of the paper is organized as follows: In Section 2, we will cover related literature on threat modeling, thereby establishing a foundational baseline for our research. Section 3 details our research methodology, including the specific search query terms used for gathering relevant literature. The findings from our literature review are presented in Section 4. Subsequently, Section 5 offers a comprehensive discussion to compare and evaluate the various automation approaches that have been presented. The paper concludes with Section 6, providing insights and perspectives on potential future research directions in the field of ATM.

2. RELATED WORK

Recent research in the field of threat modeling has been extensively explored through several review papers but is not focused on ATM. Xiong et al. (2019) offer a comprehensive overview of TM approaches, categorizing them into manual, automatic, formal, and graphical methods. Their systematic literature review, published in 2018, notes a predominance of manual approaches in the field [26]. While they acknowledge the emergence of automated methods, these mainly depend on manual processes, such as system-specific customization or manual creation of models like attack trees.

Theurich et al. discuss the challenges and practices of TM in agile environments based on a structured literature review, mentioning ATM as a promising method to reduce the high cost associated with traditional TM practices [22].

Similarly, a review focused on TM in cloud environments by Kharma and Thaweel (2023) highlights automation as a crucial element in simplifying and enhancing the productivity of the TMP [12].

3. METHODOLOGY

To examine the current state of ATM, we conducted a literature review covering the ACM Digital Library, the Web Of Science, Springer Link, IEEE Xplore and Google Scholar. The research was initiated using broad search terms, including *threat modeling* and *automation*, before refining the query to exclude results unrelated to threat modeling in the SDLC. Our query was formulated as follows:

```
BQ := ("threat modeling" OR "threat modelling")
AND ("automation" OR "automatic")
AND "software"
```

The query was adjusted to the particular syntax requirements of the search engines. Accessible results were then evaluated against a set of inclusion and exclusion criteria: The selected studies were required to be grounded in recognized threat modeling techniques, ensuring methodological soundness. We focused on works that addressed threat *modeling* in the design and development phase of the SDLC, as opposed to those centered on threat *intelligence* of operational systems, only mentioning partial aspects of threat modeling. Additionally, papers limited to a specific threat domain such as *privacy threats* were excluded to focus on automation of the general TMP.

After a paper was determined to fit these criteria, snowballing was performed on references and citations to identify additional relevant literature. Snowballing results were again evaluated against the inclusion and exclusion criteria. To capture the most recent advancements and current trends, only studies published from 2018 onwards were included. This cutoff was implied by the extensive review by Xiong et al., which provides a thorough examination of earlier literature in the field of threat modeling, including automation [26].

4. APPROACHES FOR ATM

Our investigation identified three categories of ATM approaches: machine-learning-supported, ontology-based, and configuration-based. These approaches all follow a common process to automate threat modeling. As illustrated in Figure 1, the process is structured into two phases. The initial phase involves the creation of universal artifacts applicable across multiple threat modeling iterations in various software development projects. This phase comprises two key steps: firstly, the establishment of an architecture metamodel, and secondly, the development of a threat catalog. The architecture metamodel specifies the concepts and components essential for system modeling within the chosen ATM approach. Concurrently, the threat catalog, typically tailored to the architecture metamodel's specifications, either enumerates security threats or outlines security rules to mitigate such threats. Most of the approaches we examined are equipped with a predefined metamodel and a comprehensive threat catalog.

Subsequently, the second process phase is often tailored to individual software development projects. It begins with the system's architectural modeling, which may include custom extensions to the metamodel to accommodate complex architectures. This phase also allows for the incorporation of domain-specific threats into the threat catalog. The final step involves identifying threats within the system, culminating in a compiled list of identified threats.

The structure of the process loosely resembles the three-step TMP specified by OWASP, containing the application architecture and threat identification phases [9]. Notably, none of the presented approaches focuses on mitigation techniques for identified threats. However, this was to be expected as the specification of mitigation techniques as defined by OWASP or Shostack is a task very specific to the currently modeled system [9, 20].

The following sections will detail how each of the ATM approaches implements this process, depicting the current state of research in ATM and thus answering RQ1.

4.1 Machine-Learning-Supported Generation of Threat Catalogs

Schaad and Binder [18] propose a method to automatically derive STRIDE-based threat catalogs from weakness and vulnerability databases leveraging machine learning (ML) methods. Implemented within the OVVL Threat Modeling Tool, this approach primarily contributes to the creation of universal artifacts, specifically the threat catalog, which is a key element in the first phase of the ATM process. Although the subsequent stages of the ATM process, namely system-specific architectural modeling, and threat identification, fall outside this paper's scope, they are facilitated by the OVVL tool. This tool not only assists in architectural modeling but also aids in threat identification using the generated threat catalog [18]. OVVL's architecture metamodel is not open for customization and focuses on the dataflow diagrams (DFDs) created in the tool. An overview of all artifacts of the ML-

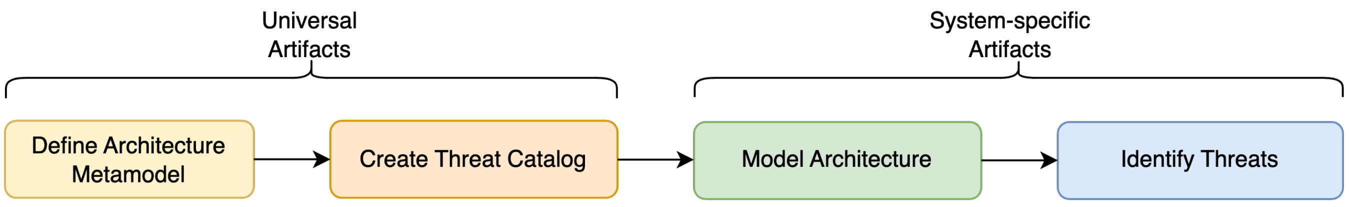

Figure 1: The ATM process

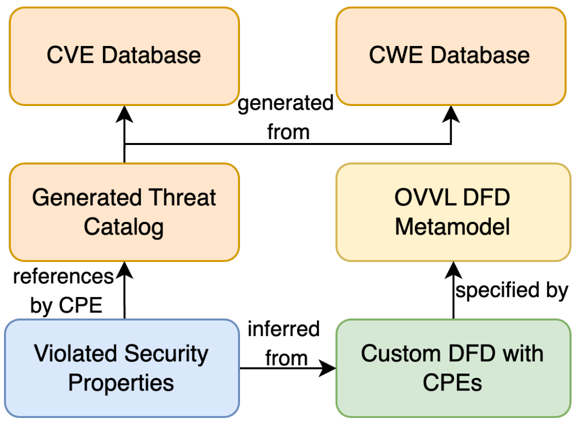

Figure 2: Artifacts of the ML-supported ATM process

supported ATM process can be found in Figure 2.

The catalog generation leverages the Common Vulnerabilities and Exposures (CVE) database [15], containing identified cybersecurity vulnerabilities for numerous software components and libraries. Each vulnerability is associated with a Common Platform Enumeration (CPE) identifier that accurately specifies the affected software or library versions [15]. These vulnerabilities needed to be mapped to valid STRIDE categories to be used in the catalog. 60 % of CVE vulnerabilities are directly associated with one or more weaknesses in the Common Weakness Enumeration (CWE) database [16]. The authors propose mapping CWE weaknesses to STRIDE categories based on the weakness scope and impact listed in each CWE entry. This makes the larger proportion of the CVE vulnerabilities directly suitable for inclusion in the threat catalog. The remaining 40 % of vulnerabilities are now mapped to STRIDE categories using a text classification model trained on the already mapped CVEs. The model yields precision and recall scores of almost 100 % on all STRIDE categories except for Repudiation, which had the lowest sample size in the training data. The resulting threat catalog has 129.675 entries. To make the catalog more manageable, the authors decided to group threats by the four categories *Access Vector* (e.g., access by network), *Authentication required*, *Programming Language* and *Technology* (e.g., web server).

After that, the threat catalog was integrated into the OVVL Threat modeling tool and comes into use when modeling the DFD of a system. Upon adding a new component to the DFD, CPEs associated with the component can be entered, and the four categories mentioned above need to be specified. Next, the tool outputs a list of CWEs and CVEs with their STRIDE categories identified by either the mapping or the ML classification. The list is sorted by severity and relevance, prioritizing severe and frequent vulnerabilities and weaknesses. After modeling the entire system, the user receives a comprehensive overview of suggestions for possible weaknesses for each system element (Violated Security Properties) [18].

4.2 Ontology-Based Approaches

Ontologies are formalized frameworks that define and represent a domain's concepts, entities, and their relationships [3]. In the context of ATM, ontologies are used to formalize software systems, enabling automation of the TMP by the inference of logic rules. These logic rules are usually formulated using description logics (DLs) like OWL and are designed to "mimic[...] the typical expert's way of thinking" [8]. Consequently, one can think of DLs as rules putting entities of the ontology in relation to each other. Special DL-reasoners can then deduct facts (e.g., a potential threat for a software component) from those rules [3]. In the following, we present three different approaches to ontology-based ATM. Figure 3 depicts the core artifacts constructed within each approach as well as the dependencies between them.

4.2.1 OdTM/Brazhuk

Traditional DFD-based architecture models hinder automated threat modeling due to their informal structure [3]. Addressing this, Brazhuk introduced the OdTM framework [3]. Following the OdTM framework, a security expert first creates a base metamodel. This metamodel is an OWL ontology that describes DFDs, formally defining their core entities, like components, data flows, trust boundaries, and their relations. It also specifies rules that make threat and countermeasure inference possible by connecting countermeasures to data flows and threats. The base metamodel also integrates widespread protocols like HTTP and DNS [4].

However, the base metamodel is not directly utilized to create the DFD components. Instead, a domain-specific ontology is derived from the base metamodel. It is essentially a domain-specific metamodel and is meant to be created and managed by a security expert who manually extends the base metamodel by domain-specific architecture components, threats, and countermeasures while maintaining a connection to the entities in the base metamodel. For his domain-specific metamodel, Brazhuk includes attack patterns from the CAPEC (Common Attack Pattern Enumerations and Classifications) [1] dictionary and weaknesses from CWE that aid in creating the domain-specific metamodel.

[1]`https://capec.mitre.org/`, Accessed: 17.12.2023

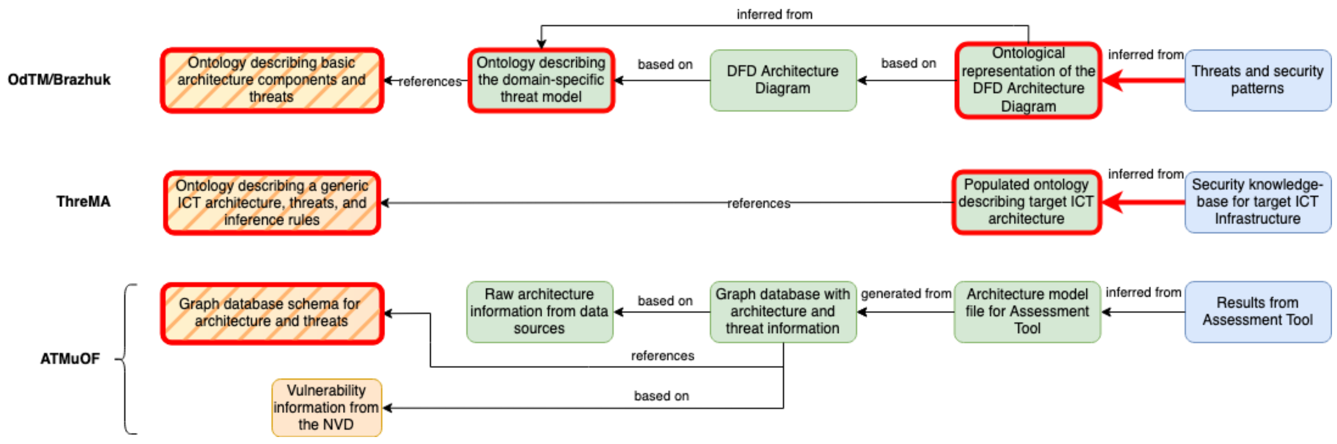
inferred from
OdTM/Brazhuk
Ontology describing basic architecture components and threats
references
Ontology describing the domain-specific threat model
based on
DFD Architecture Diagram
based on
Ontological representation of the DFD Architecture Diagram
inferred from
Threats and security patterns
ThreMA
Ontology describing a generic ICT architecture, threats, and inference rules
references
Populated ontology describing target ICT architecture
inferred from
Security knowledge-base for target ICT Infrastructure
ATMuOF
Graph database schema for architecture and threats
Raw architecture information from data sources
based on
Graph database with architecture and threat information
generated from
Architecture model file for Assessment Tool
inferred from
Results from Assessment Tool
references
Vulnerability information from the NVD
based on

Figure 3: Displayed artifacts from the ontology-based ATM approaches, color-coded by ATM process steps. Orange stripes indicate a combination of metamodel and threat catalog. Red borders indicate artifacts linked to ontologies or derivation steps resulting from ontology-inference procedures.

The first implementation of the OdTM-framework was also created by Brazhuk [1]. It focuses solely on cloud applications and attempts to address and ease the manual process of constructing domain-specific models by providing a catalog of security patterns. This catalog is derived from sources like the ENISA (European Union Agency for Cybersecurity) or OWASP and outlines common security patterns that follow a fixed format described in previous work [3]. It aims to provide more domain-scoped components for building cloud-based systems and, based on these components, supply the modeler with additional domain-specific threats. To enable non-experts to use the catalog, the paper contributes illustrations of common architectural structures and threats in cloud-based applications. The author does not provide any form of evaluation of his work.

4.2.2 *ThreMA*

De Rosa et al. present ThreMA, another ontology-based approach similar to the OdTM framework [8]. In contrast to OdTM, the authors of ThreMa contribute a more precise description of applying the framework and involved stakeholders. Moreover, they also provide a case study from a real-world use case, indicating the applicability of their approach.

ThreMA can be seen as a procedure that requires two inputs: a metamodel (seen on the left of Figure 3 in yellow-orange) and the concrete architecture model that references the latter (seen in the middle of Figure 3 in green).

The metamodel is an ontology describing generic components for architecture modeling. It is maintained by a dedicated metamodel maintainer. Similar to the base metamodel in the OdTM framework, it also contains inference rules that connect the architecture components to threats, enabling threat deduction. Based on this metamodel, an infrastructure architect manually creates a DFD-based infrastructure model to reflect the domain-specific architecture components and their communication flows. After this manual step, the infrastructure model and the threats provided by the metamodel are used to infer the applicable threats automatically. Then, the inference result needs to be analyzed by a security architect. Finally, a report of all threats identified as relevant is manually created [8].

ThreMA has been applied in a real-world system with around 100 components and data flows in the Italian public sector. The study revealed that ThreMA could successfully infer and classify more than 1000 possible threats affecting this particular system [8].

4.2.3 *ATMuOF*

Välja et al. presented an **ATM** approach that **u**ses an **o**ntology **f**ramework (ATMuOF) to normalize the names of architecture components and ensure a common abstraction level, addressing data quality issues identified in other Threat Modeling (TM) approaches [23].

The authors first model an ontology using content and reasoning patterns, merged and implemented through a graph database schema (seen on the left of Figure 3 in yellow-orange). Therefore, the patterns describe the desired structure (content patterns) of the later-ingested real-world data from different data sources reflecting the system architecture. The graph database schema will be enforced by normalization procedures (reasoning patterns) discussed later to leverage a database with a strong data schema.

Information on system architecture is sourced from various tools, including host machine commands for software lists, network scanners, and programs capturing data flows, operating system names, and vulnerability data from the national vulnerability database. Adapters specific to these sources process the data using the ontology framework's functions, predefined by reasoning patterns, and query a graph database for tasks like vulnerability classification. While the method for sourcing data, like application names or data flows, is detailed in another study [13], the ontology framework's functions are standalone and can be integrated into other environments. Consequently, if an architecture description already exists, deploying the aforementioned sourcing solution is unnecessary [23].

The normalized architecture data is then combined into a model file, which encapsulates a full description of the architecture. Finally, the model file is loaded into SecuriCAD, a tool enabling system analysis and threat modeling.[2]

[2]SecuriCAD is no longer available; most likely because the maintaining company, Foreseeti, has been

In a case study involving three enterprise system setups, the ontology framework could demonstrate that the offered functions could normalize operating system names, application software names, and the classification of application software. As with the other ontology approaches, this work also lacks formal metrics that could give insight into the quantity and quality of the discovered threats.

4.3 Configuration-based ATM

Configuration-based approaches implement the ATM process by utilizing existing Infrastructure as Code (IaC) artifacts of modern software development, such as Helm charts and CloudFormation files, to derive system threats. Such artifacts configure the setup and interaction of different system components, giving a detailed overview of the software architecture [2]. The schema of IaC files serves as a blueprint for defining the architecture metamodel in the ATM process. Important cloud-native concepts such as storage buckets or message bus subscriptions are represented in the metamodel and can be instantiated later in the architecture modeling step. The threat catalog creation step for these methods is backed by industry-recognized threat catalogs such as the Microsoft Kubernetes threat matrix [25].

The following sections present three different approaches to configuration-based ATM. The first two approaches focus on applying formal techniques to IaC files, which derive a set of threats based on pre-defined rules. In the third approach, threat simulations are conducted on running environments based on test cases derived from an existing threat catalog. An overview of the artifacts of these approaches, mapped to the ATM process step, can be found in Figure 4.

4.3.1 Formal Methods

Description Logic generation.

Cauli et al. have proposed a method of translating IaC files into a format compatible with formal reasoning, with an aim to detect design flaws and threats based on system architecture [6]. Developed specifically for CloudFormation, an Amazon Web Services (AWS) proprietary IaC standard, the approach seeks to assess system architecture in relation to a set of security properties. A metamodel for CloudFormation files is created using description logic to enable this, thereby allowing the translation of CloudFormation files into a formal language suitable for machine-processed security analysis.

The metamodel defines the concepts for all CloudFormation components, such as an S3 bucket. Leveraging description logic, the S3 bucket concept captures attributes and associated relationships, including its name and affiliated buckets used for logging actions performed on it. As with ontologies, this formal definition of components allows for automatic evaluation of its configuration in compliance with security standards. These standards are expressed as security properties also using description logic and can be assessed against all components. Security properties either specify that protection for a threat *must* be present or that an issue or threat *may* be induced by the system. An example is the *must*-security property "All S3::Buckets *must* be encrypted". The example was extracted from the tool's GitHub repository [5]. More complex properties such as the *may*-property "There *may* be a networking component that opens all ports to all" are also definable using the description logic.

During the architecture modeling step, entries of CloudFormation files are automatically encoded into description logic instances. No manual involvement is necessary. In the next step, a satisfying instance solver is used to evaluate the security properties against the description logic metamodel, the modeled system architecture, and the security properties. It returns satisfied security properties and can also return concrete instances (e.g., S3 buckets) fulfilling these properties. The result will show which of the *must*-properties are unsatisfied, indicating a security risk. *May*-properties can be unsatisfied or have zero or more satisfying instances. For example, all instances satisfying the aforementioned *may*-property checking for open ports have unprotected open ports.

The affected components are collected, and a final report based on the logic-solving results is generated for the user. The entire process runs automated on user-defined CloudFormation files with a customizable pre-provided catalog of security properties and terminological knowledge. According to the authors, an average CloudFormation file consists of 50-100 resources and finishes checking against up to 100 security properties in less than 20 seconds [6].

To cover even more threats, the authors propose a strategy to extend the formalized IaC to model data flows using domain-specific ontologies similar to those described in the previous section. This allows the integration of data flow analysis into the existing framework, thus enhancing the threat modeling capabilities. The authors used domain-specific DL axioms describing the AWS services to model the data flows. Since there can be multiple CloudFormation files for different environments, the trust boundaries can also be modeled, making it possible to create security properties checking data flows over trust boundaries, augmented by the modeling power described previously. Additionally, this modeling approach allows for the automatic generation of data flow diagrams, supporting further threat modeling activities [6].

Attack tree generation.

Blaise and Rebecchi present a different formal approach to ATM using IaC files. Instead of encoding CloudFormation files into description logic, they offer an algorithm to automatically derive attack trees from Helm charts used in Kubernetes (k8s) deployments [2]. Attack trees are an alternative to STRIDE, modeling the path of attacking a system from the entry point (e.g., a REST Endpoint) to reach a specific goal (e.g., access to customer data), with each path from root to leaf node representing a different attack path [20]. Their approach leverages the threat matrix for Kubernetes by Microsoft [25, 24], and several best practice resources on designing secure k8s deployments [2] to create the threat catalog.

Before generating the attack trees, a graph modeling the system is derived from the Helm charts. The graph's nodes represent system resources, including actual components of the application, secrets, networks, volumes, and application entry points. Edges between these nodes represent connections between resources. In the next step, the graph is enriched with information on access paths, network policies,

acquired by Alphabet `https://cloud.google.com/blog/products/identity-security/introducing-new-capabilities-for-secure-transformations?hl=en`, Accessed: 17.12.2023

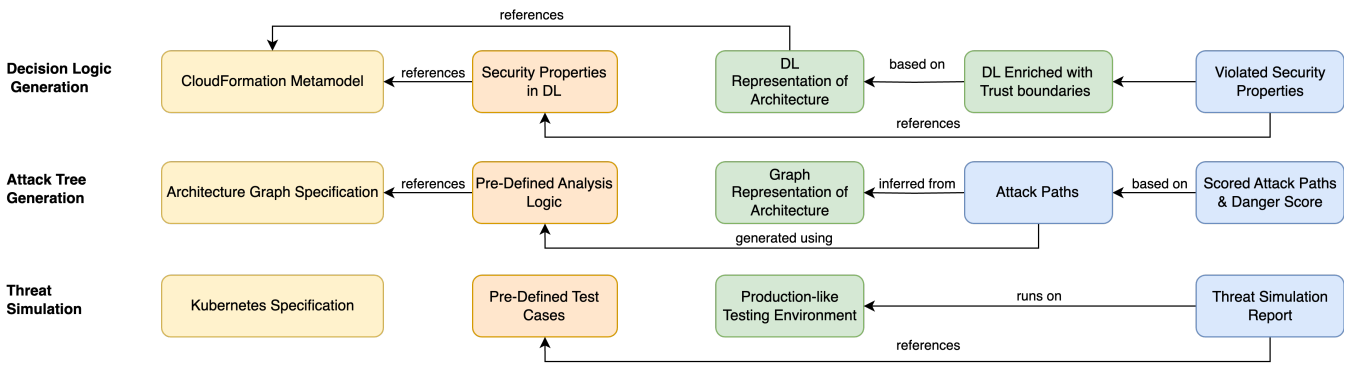
Decision Logic Generation
Attack Tree Generation
Threat Simulation
CloudFormation Metamodel
references
Security Properties in DL
DL Representation of Architecture
based on
DL Enriched with Trust boundaries
Violated Security Properties
references
references
Architecture Graph Specification
references
Pre-Defined Analysis Logic
Graph Representation of Architecture
inferred from
Attack Paths
based on
Scored Attack Paths & Danger Score
generated using
Kubernetes Specification
Pre-Defined Test Cases
Production-like Testing Environment
runs on
Threat Simulation Report
references

Figure 4: Artifacts of the Configuration-Based ATM approaches. Color-coded according to the ATM process steps.

and interaction between the components. After that, all resources are scored based on their criticality during attacks. This scoring is used while extracting attack paths from the graphs. The extraction uses a heuristic, starting at a system entry point and exploring connected resources. Finally, the tool outputs the paths in an attack tree and calculates a global danger score. After the analysis, users can also evaluate specific attack paths[2]. In an evaluation using public Helm charts for frequently used projects, the authors identified a median number of 100 critical vulnerabilities per chart, underlining the impact of their tool.

4.3.2 *Threat Simulation*

Threat Simulation is a vulnerability identification approach described by Massoud [14]. In contrast to the formal approaches described above, test cases are executed in a running environment to identify possible security risks. The technique is again focused on Kubernetes environments. This approach replaces the threat catalog with a set of test cases. Like Blaise and Rebecchi, Massoud identified these test cases using the Microsoft Threat Matrix for Kubernetes, covering topics like resource hijack attacks, data destruction attacks, or denial of service attacks. The tests can be run against a Kubernetes deployment of the actual system using a custom-made tool. The health probes of the Kubernetes pods are used to determine if an attack affects a system and may indicate elevated error levels or high response times in case an attack affects the component. Test results need to be evaluated manually, and the tests are not customized to certain pod types, but run on all pods regardless of its type [14]. The manually-created evaluation report then contains a list of identified threats and vulnerabilities. Because the test cases are created based on the Microsoft Threat Matrix for Kubernetes, the identified threats can then be used as artifacts for the mitigation stage of the TMP.

5. DISCUSSION

To enable a formal comparison between the presented approaches and to answer RQ2, we conduct a systematic assessment based on the ATM process depicted in Figure 1. The findings, categorized in the dimensions *level of automation* and *expert involvement* are used to estimate the approaches' potential industry applicability.

Since this review paper is focused on methods to automate the TMP, we included the degree of automation of each presented approach in the ATM process steps as an evaluation dimension. The scale for this dimension was chosen as follows:

- **Manual**: Artifact creation is conducted by hand
- **Semi-Automated**: Artifact creation is accelerated through specialized tools
- **Automated**: Artifact creation requires no manual actions

Another dimension that heavily influences the chance of adaptability of an approach is its need for expert knowledge and experience. In this discussion, we define an expert as someone with a large amount of knowledge in an area specific to security or the technology used in an approach (e.g., ontologies, DLs, or IaC artifacts). For this dimension, we also choose a 3-point scale:

- **No Involvement**: This step can be executed without consulting an expert
- **Partial Involvement**: This step is assisted by pre-provided expert-created resources or needs manual adaption by an expert
- **Full Involvement**: The step is completely dependent on an expert

Our assessment of both dimensions can be found in Table 1 and 2. As mentioned above, in the following, we will comment on our assessments for each step of the ATM process and link our findings to the industry applicability of the ATM approaches.

Define Architecture Metamodel.

The most prominent similarity between all approaches is the fully manual definition of the respective architecture metamodels, like the metamodel for DFDs or the formal definition and integration of threats and vulnerabilities. Additionally, experts are fully involved during this step, as creating the metamodel requires deep knowledge of the system's technology stack and the chosen ATM approach. However, as mentioned before, many systems share common deployment environments such as a specific IaC language, domain characteristics such as IoT, or frequently-used protocols like HTTP, which enables the reuse of the metamodels between different projects. Therefore, we believe an open-source environment would be ideal for industry users of these approaches to share and adapt their architecture metamod-

Approach	**Define Architecture Metamodel**	**Create Threat Catalog**	**Model Architecture**	**Identify Threats**
ML-Catalogue Generation	Manual	Semi-Automated	Manual	Manual
Ontology-Based	Manual	Manual	Manual	Automatic
Configuration-Based	Manual	Manual	Automatic	Automatic

Table 1: Comparison of automation in presented ATM approaches

Approach	**Define Architecture Metamodel**	**Create Threat Catalog**	**Model Architecture**	**Identify Threats**
ML-Catalogue Generation	Full Involvement	Partial Involvement	Partial Involvement	Partial Involvement
Ontology-Based	Full Involvement	Partial Involvement	Partial Involvement	No Involvement
Configuration-Based	Full Involvement	Partial Involvement	No Involvement	No Involvement

Table 2: Comparison of expert involvement in the presented ATM approaches

els effectively. An active community dedicated to maintaining metamodels could also increase the adoption rate of the ATM approaches.

Create Threat Catalog.

Creating threat catalogs is also a predominantly manual step in the presented approaches. Users must manually aggregate threats and/or vulnerabilities relevant to the system they are working on. This requires the involvement of security experts who can identify and categorize possible threats and add them to the catalog. This will probably hinder industry adoption as skilled staff is scarce and expensive at the time of writing. The ML-supported catalog generation approach presents a special case where the classification of threats is automated based on a manually created mapping. As such, most of the threat catalog creation step is automated, leading to a semi-automated classification. The initial mapping of categories requires expert involvement, but later iterations of classification can rely on the previously created mapping. Similar to the above step, artifacts of the threat catalog creation are suitable for sharing between different projects, easing the industry adoption challenges.

Model Architecture.

Architecture modeling is only automated by the configuration-based approaches. All the other approaches depend on manual modeling with DFDs. This is crucial, as human errors could prevent the detection of threats. Consequently, weaknesses generally associated with the practices of whiteboard hacking-based TM remain part of the ontology-based and ML-supported ATM approaches. Whether organizations are willing to pay the overall price of adoption with those persisting weaknesses in the TMP is questionable. The automatic modeling of the architecture based on IaC files mitigates this error by providing a single source of truth for the system model: the configuration of the actual production system. This removes the effort of manually remodeling the system and greatly eases the adaptability burden for configuration-based ATM approaches. Additionally, the involvement of experts in this step is not necessary since the modeling is fully automated. For the other approaches, expert involvement in modeling the data flows and trust boundaries is generally advisable to reduce human error, especially when newly implementing threat modeling.

Identify Threats.

Threat identification is mainly automatic in the ontology-based and configuration-based approaches. Ontologies leverage their inference capabilities through DLs, and the configuration-based methods use either a DL reasoner or custom logic to identify the threats. Additionally, expert involvement is not required for these approaches, as the threat catalogs are automatically generated, and the inference of threats is fully automated, not requiring expert oversight. The only exception to this is the threat simulation approach presented in this paper. However, it is still in its infancy and requires future research to enable automatic identification of threats. The presented ML-supported catalog generation approach, on the contrary, is a manual process, as the corresponding paper only focuses on automating catalog generation. Thus, the adaptability of this approach is dependent on future work that describes how one can adapt this technique for all steps of the TMP.

Accuracy of Approaches.

The accuracy of threat modeling approaches is critical, especially in the context of discovered threats. Unfortunately, all discussed approaches are still in an early stage of development, and the presented literature lacks scientific evaluations focusing specifically on the accuracy of these approaches.

6. CONCLUSION

We presented and evaluated ATM approaches in the configuration-, ontology-based, and ML-supported catalog generation categories. Based on our findings, there is no "one-fits-all" approach applicable to current software development projects yet, and all ATM approaches are still in their early stages, with no empirical studies conducted to evaluate ATM in the industry and check the result accuracy.

While the current research already shows significant potential to increase the adoption and degree of automation of threat modeling in the industry, further work needs to be done to enable the generalized application of the presented methods. Future research could focus on combining the listed approaches into a fully automated ATM process, leveraging ML-generated threat catalogs and IaC-based architecture model generation to infer threats using a large ontology.

Additionally, a study could focus on integrating threat simulation as a verification step to the other approaches, ensuring accurate TM results once the threat simulation evaluation is automated. Future work needs to focus on sharing metamodels as open-source artifacts between projects to reduce metamodel creation effort and make ATM easy to integrate into modern SDLCs.

7. REFERENCES

[1] B. Andrei. Threat modeling of cloud systems with ontological security pattern catalog. *International Journal of Open Information Technologies*, 9(5):36–41, 2021.

[2] A. Blaise and F. Rebecchi. Stay at the Helm: secure Kubernetes deployments via graph generation and attack reconstruction. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 59–69, Barcelona, Spain, July 2022. IEEE.

[3] A. Brazhuk. Security patterns based approach to automatically select mitigations in ontology-driven threat modelling. 2020.

[4] A. Brazhuk. Owasp ontology-driven threat modelling (odtm) framework. `https://github.com/nets4geeks/OdTM/blob/master/OdTMBaseThreatModel.owl`, 2021. Accessed: 25.11.2023.

[5] C. Cauli. CloudFORMAL: Reasoning properties for aws s3. `https://github.com/claudiacauli/CloudFORMAL/blob/150a201e43e8dac06cdbe0f475837ee48e652364/src/main/scala/com/cloud/formal/reasoning/properties/s3.json`, 2021. Accessed: 25.11.2023.

[6] C. Cauli, M. Li, N. Piterman, and O. Tkachuk. Pre-deployment Security Assessment for Cloud Services Through Semantic Reasoning. In A. Silva and K. R. M. Leino, editors, *Computer Aided Verification*, volume 12759, pages 767–780. Springer International Publishing, Cham, 2021. Series Title: Lecture Notes in Computer Science.

[7] L. Conklin, D. Victoria, and S. Strittmatter. Threat Modeling Process. OWASP Foundation.

[8] F. De Rosa, N. Maunero, P. Prinetto, F. Talentino, and M. Trussoni. ThreMA: Ontology-Based Automated Threat Modeling for ICT Infrastructures. *IEEE Access*, 10:116514–116526, 2022.

[9] O. Foundation. OWASP Ontology Driven Threat Modeling Framework.

[10] A.-M. Jamil, S. Khan, J. K. Lee, and L. Ben Othmane. Towards Automated Threat Modeling of Cyber-Physical Systems. In *2021 International Conference on Software Engineering & Computer Systems and 4th International Conference on Computational Science and Information Management (ICSECS-ICOCSIM)*, pages 614–619, Pekan, Malaysia, Aug. 2021. IEEE.

[11] M. Kharma and A. Taweel. Threat Modeling in Cloud Computing - A Literature Review. In G. Wang, K.-K. R. Choo, J. Wu, and E. Damiani, editors, *Ubiquitous Security*, volume 1768, pages 279–291. Springer Nature Singapore, Singapore, 2023. Series Title: Communications in Computer and Information Science.

[12] M. Kharma and A. Taweel. Threat Modeling in Cloud Computing - A Literature Review. In G. Wang, K.-K. R. Choo, J. Wu, and E. Damiani, editors, *Ubiquitous Security*, pages 279–291, Singapore, 2023. Springer Nature Singapore.

[13] KTH Royal Institute of Technology, 100 44 Stockholm, Sweden, M. Välja, R. Lagerström, KTH Royal Institute of Technology, 100 44 Stockholm, Sweden, U. Franke, RISE Research Institutes of Sweden, 164 40 Kista, Sweden, G. Ericsson, and KTH Royal Institute of Technology, 100 44 Stockholm, Sweden. A Framework for Automatic IT Architecture Modeling: Applying Truth Discovery. *Complex Systems Informatics and Modeling Quarterly*, (20):20–56, Oct. 2019.

[14] A. Massoud. Threat Simulations of Cloud-Native Telecom Applications. Master's thesis, Aalto University. School of Electrical Engineering, 2021.

[15] MITRE Corporation. Common vulnerabilities and exposures (cve), 2023. Accessed: 2023-11-26.

[16] MITRE Corporation. Common weakness enumeration, 2023. Accessed: 2023-11-26.

[17] MITRE Corporation. Mitre att&ck, 2023. Accessed: 2023-11-26.

[18] A. Schaad and D. Binder. ML-Supported Identification and Prioritization of Threats in the OVVL Threat Modelling Tool. In A. Singhal and J. Vaidya, editors, *Data and Applications Security and Privacy XXXIV*, pages 274–285, Cham, 2020. Springer International Publishing.

[19] A. Shostack. Experiences Threat Modeling at Microsoft. page 35, 2008.

[20] A. Shostack. *Threat modeling: designing for security.* Wiley, Indianapolis, Ind, 2014.

[21] L. Sion, K. Yskout, D. Van Landuyt, A. Van Den Berghe, and W. Joosen. Security Threat Modeling: Are Data Flow Diagrams Enough? In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 254–257, Seoul Republic of Korea, June 2020. ACM.

[22] P. Theurich, J. Witt, and S. Richter. Practices and challenges of threat modelling in agile environments. *Informatik Spektrum*, 46(4):220–229, Aug. 2023.

[23] M. Välja, F. Heiding, U. Franke, and R. Lagerström. Automating threat modeling using an ontology framework: Validated with data from critical infrastructures. *Cybersecurity*, 3(1):19, Dec. 2020.

[24] Y. Weizman. Threat matrix for Kubernetes, 2020. Publisher: Microsoft Defender for Cloud.

[25] Y. Weizman. Secure containerized environments with updated threat matrix for Kubernetes, 2021. Publisher: Microsoft Defender for Cloud.

[26] W. Xiong and R. Lagerström. Threat modeling – A systematic literature review. *Computers & Security*, 84:53–69, July 2019.

[27] K. Yskout, T. Heyman, D. Van Landuyt, L. Sion, K. Wuyts, and W. Joosen. Threat modeling: from infancy to maturity. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 9–12, 2020.

Towards a Comprehensive Model for Classifying Software Vulnerabilities and Countermeasures

Patrick Treppmann
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
patrick.treppmann@rwth-aachen.de

Florian Braun
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
florian.maximilian.braun@rwth-aachen.de

ABSTRACT

Classification of software vulnerabilities allows their identification and mitigation to be effectively carried out at any time. While various classifications exist, a holistic model for vulnerability classification across different perspectives is currently unavailable. Our research reviews existing classifications from literature and merges important aspects into a comprehensive model. The resulting model enables the identification of vulnerabilities from various perspectives, including threat, attack surface, development phase, security mechanism and mitigation strategy. A mapping of existing vulnerability data, such as CWE entries, to the model could facilitate an effective lookup of relevant information, to support all kinds of software security tasks.

Keywords

Software Security, Vulnerability, Meta-Model

1. INTRODUCTION

Classifications of software vulnerabilities provide a valuable support in enhancing software security and mitigating vulnerabilities [7] [6].
We define the term *vulnerability* to avoid ambiguities by using the following NIST[1] definition:

> Weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source [10].

Existing classifications generally consider vulnerabilities from one specific perspective such as attack surface [9], architectural weaknesses [11] or implementation errors [12]. Thus there is a lack of a comprehensive, unified model that integrates the various specific classifications and enables a broader view on vulnerabilities. This research addresses the absence of such a model by providing a unified framework for classifying software vulnerabilities and mitigation strategies. The central objective of this work is to address the following research question:

> RQ: "How can a comprehensive model be developed to integrate diverse perspectives on software vulnerabilities?"

The outcome of this research is expected to have practical implications regarding the classification of CWE[2] (Common Weakness Enumeration) database entries, aiming to provide more utility of the database in the practice of designing, developing and maintaining secure software.
The following chapter 2.1 presents existing classifications found in literature which serve as the basis for the unified model. Chapter 2.2 briefly introduces the CWE database that is used to validate the developed model. Chapter 3 outlines the applied methodology for finding relevant literature and constructing the model. Subsequently, chapters 4.1 and 4.2 provide a comprehensive presentation of the resulting model, detailing its key elements and their interconnections. Chapter 4.3 goes into detail on which parts of existing classifications are incorporated into the unified model. Chapter 5 engages in a discussion of the results, including the implications and limitations of the model presented in chapters 5.2 and 5.3. This includes a mapping of three existing CWE entries to the model in chapter 5.1. Lastly chapter 6 provides a conclusion of the paper and an outlook on valuable future work in chapter 6.1.

2. RELATED WORK

This chapter presents existing classifications found in literature to provide an overview of the models that are incorporated as the basis of the resulting model.

2.1 Existing Classifications

The paper *Seven Pernicious Kingdoms: A Taxonomoy of Software Security Errors* introduces a framework tailored to developers, by prioritizing practicality and relevance over theoretical completeness [12]. Classification is done on a code level focusing on vulnerabilities that appear in software applications. The taxonomy comprises seven categories, ranked by their importance for secure software, including classes like `Input Validation and Representation` and `API Abuse`. It is emphasised that this model is not theoretically complete and open to change and expansion. Due

[1] `https://www.nist.gov/`

[2] `https://cwe.mitre.org/`

SWC Seminar 2023/24 RWTH Aachen University, Germany.

to the focus on providing a practical and intuitive framework to aid developers in creating secure software, the paper distinguishes itself from other more theoretical work. In addition the created taxonomy is available in the CWE database as an external mapping, currently comprising 88 weaknesses mapped to the categories presented in the paper.
Another classification already present in the CWE database are the eleven security tactics presented in the paper *A Catalog of Security Architecture Weaknesses* [11]. The main focus of this classification is on security tactics to support the creation of a secure software architecture. A first classification of architectural vulnerabilities is presented with the three types `Omission`, `Commission` and `Realization`, describing the absence of a security tactic, the choice of an inappropriate tactic and lastly a faulty implementation of a tactic. The catalog comprises eleven tactics, ranging from actor authentication and authorization to data encryption and input validation. Entries of the CWE database are mapped to one of those security tactics and additionally annotated with one of the three types. The results can be found in the CWE view `Architectural Concepts`, currently including 223 entries.
The paper *A Grounded Theory Based Approach to Characterize Software Attack Surfaces* provides a comprehensive classification of software vulnerabilities with a focus on attack surfaces [9]. Based on data collected from the CWE as well as the CVE[3] (Common Vulnerabilities and Exposures) database, attack surfaces are classified into three core classes: `Entry Point`, `Target` and `Mechanism`. Entries of each class are assigned to an abstraction level of `Code`, `Program`, `System` or `Network`. A comprehensive model for each of the three main categories is presented featuring in total 279 entry points, targets and mechanisms.
The taxonomy of the paper *A Taxonomy of Computer Program Security Flaws* covers vulnerabilities based on three key aspects: `Genesis`, `Time of Introduction` and `Location` [8]. Genesis distinguishes between vulnerabilities with malicious intent, non-malicious intent and ones that occur unintentionally. Time of Introduction differentiates between development, maintenance and operation, while the development phase is further split up into the requirement and design, source code and object code. Lastly the category location includes the classes software and hardware.

The paper *Towards Practical Cybersecurity Mapping of STRIDE and CWE – a Multi-perspective Approach* presents different mappings of the STRIDE[4] threat model to vulnerabilities in lists from the CWE database like the `OWASP Top Ten` and the `CWE Top 25 Weaknesses`[5]. The mapping between STRIDE and CWE is based on the attributes `Technical Impact` and `Scope`, that each entry is annotated with in the database. In addition, a mapping is established between the STRIDE framework and the CIA concept of `Confidentiality`, `Integrity` and `Availability`. The authors emphasize the need to further refine the classifications found in the CWE database as well as adding mappings like the ones they proposed in order to make it more applicable in security practices.
Compared to the presented existing literature, this paper takes a different approach to software vulnerability classification. While previous works have focused on specific perspectives such as attack surfaces or architectural vulnerabilities, this paper aims to bring together these different classifications into a comprehensive unified model. This approach provides a more complete representation of software vulnerabilities. By combining various classifications, the paper recognizes the complex nature of software vulnerabilities and enhances their representation. This integrated perspective differentiates this work from previous studies, offering a more thorough understanding of software vulnerabilities.

2.2 Common Weakness Enumeration (CWE)

The CWE database is an essential resource in the field of software security and vulnerability analysis. Developed and maintained by the MITRE Corporation, CWE is a standardized system for categorizing and describing common software security weaknesses. This comprehensive repository classifies a wide range of vulnerabilities, providing detailed information about their characteristics, potential impact, and recommended mitigation strategies. CWE also classifies weaknesses based on the phase (e.g. implementation, coding, design) in which they might be introduced, providing a comprehensive framework for vulnerability analysis and mitigation. CWE categorizes software weaknesses into four levels of abstraction: `Category`, `Class`, `Base` and `Variant`. Categories group entries with a common characteristic while classes contain abstract weaknesses independent of any language or technology. Base entries, which are also mostly technology agnostic, include specific detection and prevention methods. Lastly Variants are specific weaknesses that affect only a certain language or technology [4].

3. METHODOLOGY

To construct a comprehensive unified model for the classification of software vulnerabilities, an extensive literature search is performed across four common databases, including Google Scholar, Springer Link, ACM Digital Library, and IEEE Xplore. The search query was constructed as follows:

```
"software" AND
("vulnerability" OR "vulnerabilities"
OR "weakness" OR "weaknesses") AND
("classification" OR "taxonomy" OR
"categorization")
```

Results are examined and scanned for existing classifications. All papers describing a vulnerability model are collected and analyzed to work out their commonalities, differences and connections. The focus lies on identifying different classifications that cover a wide range of diverse perspectives on software vulnerabilities like attack surfaces, security mechanisms and others. Models present in literature are selected based on their adoption in the research and practical community. This also includes classifications that are already implemented in the CWE database. Additionally models are selected that cover a unique approach or perspective which is distinct from other existing models.
In order to design a unified model, shared characteristics and elements between existing classifications are examined. To prevent ambiguity and duplication of concepts and terms, a uniform terminology is agreed upon and models are adapted accordingly. The main focus while combining existing models is to incorporate various different aspects and perspectives on software vulnerabilities to create a comprehensive

[3]`https://cve.mitre.org/`
[4]`https://learn.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20)`

Vulnerability	Phase	Requirements
		Planning
		Design
		Implementation
		Testing
		Deployment
		Maintenance
	Security Mechanism	Data Encryption
		Manage User Session
		Authentication
		Authorization
		Limit Access
		Limit Exposure
		Input Validation
		...
	Attack Surface	Network
		Software/ Code
		System
		Program
	Threat	Tampering
		Repudiation
		Spoofing
		Information Disclosure
		Denial of Service
		Elevation of Privilege
	Mitigation	Dynamic Application Security Testing
		Penetration Testing
		Libraries/ Frameworks
		Input Validation
		Least Privilege Principle
		Use Security Best Practices
		Encryption
		Dependency Scanning
		...

Table 1: Visualization of the unified model

and expressive unified model.
Due to the interconnected nature of security perspectives, the inderdependencies between different perspectives are also addressed. This ensures that the resulting model offers a comprehensive view without redundancy.

4. RESULTS

The resulting unified model of this work, depicted in table 1, classifies vulnerabilities from various perspectives. A perspective in this context is a unique lens or viewpoint through which the vulnerabilities are analyzed and categorized. It serves as a conceptual framework that enables a comprehensive understanding of the diverse aspects and nuances of vulnerabilities. Each perspective encapsulates a specific attribute of a software vulnerability, thus when combined, providing a multi-faceted view that contributes to a more nuanced and thorough vulnerability assessment. These perspectives form the first layer of the developed model and include `development phase`, `mitigation strategy`, `threat`, `security mechanism`, and `attack surface`. In the following an extensive description of these perspectives, their interrelationships and theoretical foundations is presented.

4.1 Model Description

The `development phase` perspective in the model is crucial as it helps to identify at which stage of the development process a vulnerability might be introduced. This could range from the requirements phase, where a lack of clarity or specificity could lead to potential security gaps, over the implementation phase, where bugs are introduced, to the maintenance phase, where outdated components or unpatched systems could expose vulnerabilities. Understanding the phase of introduction not only helps in pinpointing the origin of the vulnerability but also aids in devising effective mitigation strategies tailored to each specific phase.
The `mitigation` perspective is about proactively addressing vulnerabilities. It involves strategies like Dynamic Application Security Testing (DAST), which analyzes running applications for potential security threats, or the use of established libraries and frameworks that have been tested and verified for security. Mitigation also includes rectifying vulnerabilities once identified. This involves actions like software updates, configuration changes, or code modifications. Each mitigation strategy is mapped to a development phase, indicating when it would be most effective to apply.
From the `threat` perspective, vulnerabilities are categorized based on the actual threat they enable. For instance, tampering refers to unauthorized changes that could alter the

Attack Surface	Code	Weak Encryption
		Error Handling
		Improper Input Validation
		Serialization/ Deserialization
		Unhandled Race Condition
		...
	Network	Unsafe Channel/ Protocol
		Remote Access Control
		Open Port
		Type of Network Package
		...
	Program	Interaction with other Applications
		Connecting to other Servers
		User Input
		Device Input
		Missing Maintenance Updates
		...
	System	Improper Server Configuration
		Improper Access Control
		External Connection Requests
		Installing External Programs
		...

Table 2: Specification of Attack Surfaces for the *Code, Network, Program* and *System* level

system's behavior or data, while information disclosure involves the exposure of information to individuals who are not supposed to have access to it. A mapping of these threats to the CIA triad (`Confidentiality`, `Integrity`, and `Availability`) proposed by Honkaranta et al., provides a clear understanding of the potential impact of each threat [5]. Spoofing, Information Disclosure, and Elevation of Privilege directly threaten confidentiality. Integrity risks involve Spoofing, Tampering, Repudiation, and Elevation of Privilege. Spoofing and Tampering compromise data integrity, while Repudiation and Elevation of Privilege pose threats to system integrity. Availability is targeted by Denial of Service [5].
The `security mechanism` perspective pertains to the distinct security tactic that a vulnerability is associated with. This could encompass mechanisms such as data encryption, authentication, authorization processes, and others. Each category represents a specific aspect of security, providing a structured way to categorize and address vulnerabilities. More targeted and effective mitigation measures can be devised, by understanding the security mechanism related to a vulnerability.
Lastly, the `attack surface` perspective examines the potential points of entry for an attack. These could be the `network`, which could be vulnerable to attacks if not properly secured, the software or `code`, which could have bugs or errors that can be exploited, or the hardware `system` the code runs on, which could be compromised if it has weak security configurations or outdated components. Furthermore the running `program` instance could enable new attack surfaces through interaction with its environment. A representation of concrete classes for the second layer of attack surfaces was excluded from the depiction of the overall model in table 1 to avoid overloading it with too much information. Thus a more detailed classification of software attack surfaces is presented in table 2, representing types of attack surfaces for each layer two perspective. The classification in table 2 is derived from the work of Moshtari et al. by combining the three classes Entry Point, Target and Mechanism, while focusing on the abstraction levels described above [9].
By providing a comprehensive view of vulnerabilities from multiple perspectives, the model holds the potential to serve as a valuable tool for vulnerability assessment. This, in turn, may facilitate a more nuanced understanding of vulnerabilities. However, it is important to note that the actual effectiveness of the model is yet to be determined, and further validation and testing is required to assess its practical benefits in real-world scenarios.

4.2 Interconnections in the Model

In the following, a detailed exploration of the interconnections between each individual layer one perspective is presented, providing a comprehensive understanding of their influence within the landscape of vulnerabilities.
In the realm of system security, the design phase serves as the bedrock for the introduction of security mechanisms. The absence, wrong choice, or inadequate implementation of these mechanisms during this phase can lead to a system that opens up broad attack surfaces and is susceptible to a wide range of possible threats. This highlights the crucial interplay between the design phase and security mechanisms, emphasizing the importance of incorporating security considerations early in the system design process.
These security mechanisms, in turn, shape the potential attack surfaces. For example, a flawed, absent, or incorrectly implemented authentication mechanism can create an attack surface, granting access to sensitive system areas without authentication. This link between security mechanisms and attack surfaces underscores the importance of correctly implementing security mechanisms to minimize potential attack surfaces.
The characteristics of these attack surfaces can dictate the mitigation strategy. For instance, a web application with

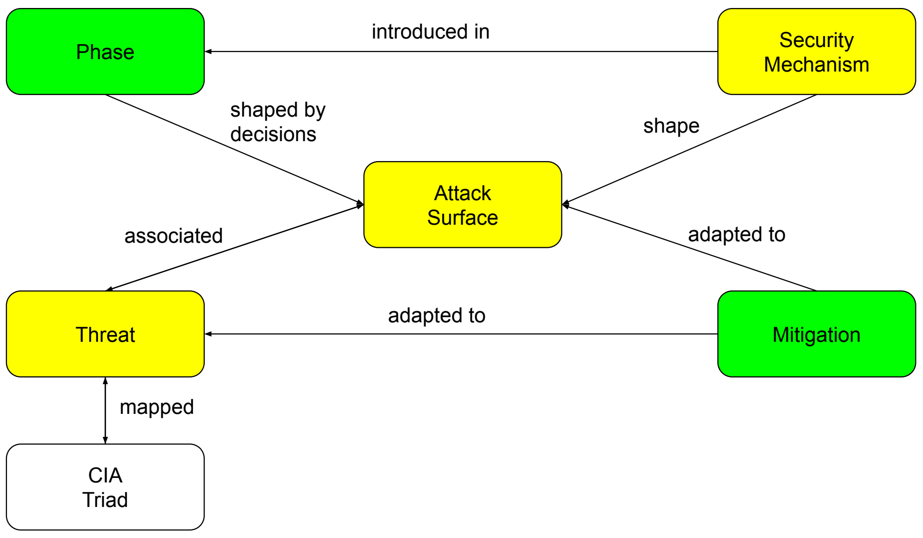

Figure 1: Interconnections of layer one perspectives in the model

an extensive attack surface necessitates mitigation strategies like deploying a firewall. This relationship between attack surfaces and mitigation demonstrates the need for tailored security measures that address the specific vulnerabilities of each attack surface.
Moreover, the choices made during the design phase and their execution in the development phase can significantly shape the existence and extent of attack surfaces. This influence of the design and development phase on the attack surface underlines the long-term impact of design and development decisions on the system's security.
Specific types of threats are associated with particular attack surfaces. For example, a `denial of service` threat or attack will likely target attack surfaces exposing system resources such as network or system memory. This correlation between threats and attack surfaces illustrates the need for a comprehensive understanding of potential threats to effectively secure all attack surfaces.
The type of threat also directly influences the corresponding mitigation strategy. For instance, the threat of `information disclosure` could potentially be triggered by weak encryption. Once this threat is identified, the mitigation strategy could involve implementing system patches to strengthen the encryption or notifying users to change their sensitive information. This connection between threat and mitigation underscores the need for a dynamic and responsive security approach that adapts to the specific threats faced by the system.
In conclusion, these interconnections offer a nuanced understanding of vulnerabilities, enabling a comprehensive and effective vulnerability assessment and management. They underscore the importance of a holistic approach to system security, where each perspective is understood not in isolation, but in relation to others. This interconnected view allows for more effective identification, analysis, and mitigation of vulnerabilities. A visual representation of these interconnections is depicted in figure 1, providing a concise overview of the relationships between various vulnerability perspectives.

4.3 Theoretical Foundations

After presenting the structure of the unified model and the interconnections between the elements, it is crucial to discuss the theoretical foundation that provided the basis for its development. The model draws on key concepts from five notable works in the field of software security. Each of these works provide insights from a unique perspective, into understanding and classifying vulnerabilities. In the remainder of this chapter it is described how these models, introduced in chapter 2.1, are incorporated and combined into a comprehensive unified model. Furthermore, an examination of which elements were not included and the reasoning behind these decisions is conducted.
The paper *Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors* introduces a taxonomy of software security errors that is tailored to developers, prioritizing practicality and relevance [12]. While the specific taxonomy is not directly incorporated into the model, its influence is evident in the layer two elements. The taxonomy's categories such as `Input Validation and Representation` and `API Abuse` resonate with the `security mechanism` and `attack surface` perspectives in layer one of the created model. These perspectives are crucial for understanding how vulnerabilities can be introduced and exploited in software applications, and how they can be mitigated. Therefore, the practical focus and emphasis on common types of coding errors in the taxonomy have significantly shaped the layer one elements of the model. This underscores the importance of practical, developer-focused considerations in the development of the model.
The influence of the paper *A Catalog of Security Architecture Weaknesses* is primarily seen in the `security mechanism` perspective of the unified model. The security mechanisms in the developed model reflect the security tactics presented in the paper. The principles and insights from the paper have significantly informed the development of the `security mechanism` layer in the model. However, the specific Common Architectural Weakness Enumeration (CAWE) catalog and the detailed classification of vulnerabilities from the paper are not directly included in the unified model. This is

because the created model aims to provide a broader perspective on software security that goes beyond specific catalogs or classifications of individual vulnerabilities.
The third paper, *A Grounded Theory Based Approach to Characterize Software Attack Surfaces* [9], provides a comprehensive classification of software vulnerabilities with a focus on attack surfaces. It classifies attack surfaces into three core classes: `Entry Point`, `Target`, and `Mechanism`, and assigns entries of each class to an abstraction level of `Code`, `Program`, `System`, or `Network`. In the developed model, the `attack surface` layer reflects the abstraction levels of Code, Program, System, and Network from the paper. These levels were chosen because they provide a broad categorization that can encompass a wide range of attack surfaces in different contexts, making the developed model versatile and applicable to various software security scenarios. However, the specific classes of Entry Point, Target, and Mechanism from the paper are not directly included in the model. This decision was made to maintain the abstraction level and clarity of the unified model. Including these specific classes would add another layer of complexity to the model, which could make it more difficult to understand and apply. By focusing on high-level categories, the created model remains accessible and user-friendly, while still providing a comprehensive overview of software vulnerabilities.
The fourth paper, *A Taxonomy of Computer Program Security Flaws*, classifies vulnerabilities based on three key aspects: `Genesis`, `Time of Introduction`, and `Location` [8]. While the unified model does not directly incorporate these specific aspects, it does reflect the paper's approach to classifying vulnerabilities. The Genesis aspect aligns with the `threat` perspective in the developed model, both focusing on the origin of vulnerabilities. The Time of Introduction aspect resonates with the `phase` perspective in our model, emphasizing the importance of considering when vulnerabilities might be introduced during the development process. The Location aspect is reflected in the `attack surface` perspective in our model, which considers where vulnerabilities might exist or be exploited. However, the specific classifications of Genesis, Time of Introduction, and Location from the paper are not directly included in the developed model. This is because our model aims to provide a more generalized framework that can accommodate various classifications and perspectives on software security.
The fifth paper, *Towards Practical Cybersecurity Mapping of STRIDE and CWE – a Multi-perspective Approach*, presents different mappings of the STRIDE threat model to existing categories of vulnerabilities included in the CWE database. While the unified model does not directly incorporate these specific STRIDE-CWE mappings, it does reflect the paper's approach to mapping threats to their corresponding mitigations in the `threat` and `mitigation` perspectives. The specific mappings between STRIDE elements and the CWE database [5] based on the attributes `Technical Impact` and `Scope` are not included in our model. However the mapping established between the STRIDE framework and the CIA concept of `Confidentiality`, `Integrity`, and `Availability` [5] are integrated into the developed model. This additional mapping is included as it provides valuable information on the impact of specific security threats. The decision to exclude the mapping of STRIDE to CWE is made to maintain the abstraction level of software perspectives and to distinguish the created model from current classifications included in the CWE database. In total, the principles and insights from the paper have significantly informed the development of the `threat` and `mitigation` layers in the unified model.
In conclusion, the created model incorporates key concepts from the five papers, including the categorization of vulnerabilities, the enumeration of architectural weaknesses, the characterization of attack surfaces, and the mapping of threats to their impact as well as the possible mitigations. This characteristic of a flexible and adaptable framework, results in a versatile model. It can accommodate a broad spectrum of software security aspects while still embodying the fundamental principles of established models. This versatility enhances its applicability across various scenarios in the realm of software security.

5. DISCUSSION

In order to test the comprehensiveness and applicability of the the presented unified model, several CWE database entries are selected through random sampling. By successfully mapping these vulnerabilities to the model, we can validate the models effectiveness and capability of categorizing known vulnerabilities within industry standards. This ensures interoperability of the model with widely accepted frameworks and easier integration into existing security ecosystems. It is important to note that the current sampling comprises only three CWE entries. While these instances serve as initial validation, it is crucial to emphasize that a more extensive mapping of vulnerabilities would be necessary to thoroughly validate and ensure the model's effectiveness across a broader spectrum of CWE entries. Ultimately, the goal is to map all CWE entries to the developed model to enable its practical use in software projects.
Each selected entry is first scanned for completeness, to avoid entries with missing or incomplete information. Required information are the description and the phase of introduction. Existing mitigations are desirable but not required. In addition, CWE entries of the type class and variant are excluded from the selection process. Class entries are described only in an abstract fashion and other entries inherit and extend their features while variant entries describe vulnerabilities that are linked to a specific product, being to detailed for a validation. This exclusion was necessary to ensure that the entries selected for the mapping provided concrete and specific information about the described vulnerability. Three CWE entries were selected and thoroughly examined.

5.1 CWE Mapping Test

The first examined CWE entry is `Incorrect Implementation of Authentication Algorithm (CWE-303)` [1]. The introduction phase can directly be mapped to the model and is classified as implementation. Mitigation strategy in this entry is missing and thus can not be mapped. The three other categories `threat`, `security mechanism` and `attack surface` are not directly contained in CWE entries and thus can only be inferred from the textual descriptions of existing categories. For `security mechanism` authentication is evident, from the description information disclosure and elevation of privilege can be inferred as threats. By bypassing authentication mechanisms an attacker could get access to information or other resources they are not authorized to ac-

cess. As the entry refers to an incorrect implementation the `attack surface` is classified as code. Lastly as mitigation strategies are not included as a category, and not mentioned in the description of the CWE entry they can not be directly mapped and it might not be possible to semantically derive them from the entry, as there is only a very short description available. However they could still be inferred by individuals with expertise in the field. A potential mitigation strategy for this entry could be the use of established authentication libraries.
The second examined CWE entry is `Improper Neutralization of Input During Web Page Generation (Cross-site Scripting)` (CWE-79) [3]. The introduction phase can directly be mapped to the unified model and is classified as implementation. Mitigation strategies in this entry are available as a CWE category and can be mapped accordingly. They include among others input validation during the implementation phase and libraries and frameworks in the design phase. The three other categories `threat`, `security mechanism`, and `attack surface` are inferred from the textual descriptions of existing categories. For `security mechanism`, input validation is evident from the mitigation strategies. From the description, tampering can be inferred as a `threat`. An attacker could manipulate the website content or even the website itself by injecting malicious scripts. As the entry refers to an improper neutralization of input during web page generation, the `attack surface` is classified as code.
The third and last examined CWE entry is `URL Redirection to Untrusted Site (Open Redirect) (CWE-601)` [2]. The introduction phase can directly be mapped to the model and is classified as both architecture and design, and implementation. Mitigation strategies in this entry are explicitly mentioned and can be mapped accordingly. They include input validation during the implementation phase, firewall during the operation phase, and attack surface reduction during both the architecture and design, and implementation phases. The three other categories `threat`, `security mechanism`, and `attack surface` are again inferred from the textual descriptions. The involved `security mechanism` is input validation, because a server redirects to a user-controlled input link. Since the vulnerability simplifies a phishing attack, the `threat` can be considered as spoofing. The `attack surface` is code and more specifically improper input validation.
In the process of mapping three CWE entries to the model, it became evident that certain categories could be directly mapped, while others required semantic analysis. Specifically, the `phase of introduction` and `mitigation strategy` categories were directly represented in the CWE entries, allowing for a straightforward mapping. However, the categories `threat`, `security mechanism`, and `attack surface` were not explicitly stated in the CWE entries. These categories required a deeper semantic analysis of the textual descriptions provided in the entries, which could be an important direction for future work. These properties are also visualized in table 1 as green, for direct mapping, and yellow for mapping through semantic analysis.

5.2 Implications

The primary goal of the unified model is to provide a structured framework for classifying software vulnerabilities. This classification system could potentially facilitate a more systematic understanding and management of software vulnerabilities. By categorizing vulnerabilities according to specific criteria such as `phase`, `mitigation strategy`, `threat`, `security mechanism`, and `attack surface`, the model could help in identifying patterns, trends, and correlations among different types of vulnerabilities. This could, in turn, support the development of more effective strategies for vulnerability prevention, detection, and mitigation. The mapping of the developed model to the three selected CWE entries serves as an initial test of the models potential for comprehensiveness and applicability in representing real-world software vulnerabilities. While this small-scale test was successful, it does not constitute a full validation of the model. However, the results are promising and suggest that, with further testing and refinement, the created model could become a valuable tool in the field of software-security. Potential applications could include vulnerability assessment, risk management, and security training. It is important to note that a more extensive validation using a larger and more diverse set of CWE entries would provide a more robust evaluation of the models effectiveness.

5.3 Limitations

While the created model provides a valuable framework for classifying software vulnerabilities, it also has certain limitations. Firstly, the effectiveness of the model may vary depending on the nature and complexity of the vulnerabilities being classified. Some vulnerabilities may not fit neatly into the existing categories of the model or may span multiple categories, which could complicate the classification process. Secondly, the model relies on certain assumptions, such as the availability of complete and accurate information about the vulnerabilities. In reality, this information may not always be available or reliable, which could limit the applicability of the model. Lastly, the model currently does not account for the dynamic nature of software security, where new types of vulnerabilities can emerge that may not fit neatly into the existing categories of the model. This highlights the need for continuous refinement and updating of the model to keep pace with the evolving software security landscape.

6. CONCLUSION

In this work we followed the research objective to implement a comprehensive model for the classification of software vulnerabilities. The resulting model provides a comprehensive framework for understanding and assessing vulnerabilities from multiple perspectives: development phase, mitigation strategies, threats, security mechanisms and attack surfaces.
The unified model was created by scanning and analyzing existing classifications, understanding their limitations, finding their commonalities and merge important aspects together into a new, holistic model. Furthermore we analyzed the relations across the different perspectives.
In the resulting model, a vulnerability is classified from every perspective. During a specific `development phase`, a vulnerability has the potential to be introduced. `mitigation strategies` describe actions to be taken to prevent the vulnerability in the first place or to reduce the potential harm caused by it. The mitigation strategies are related to development phases they can be applied in. `threats` are categorized based on the actual threat they enable, such as

tampering, spoofing, information disclosure, denial of service, repudiation, and elevation of privilege. Each threat is mapped to an element within the CIA triad (confidentiality, integrity, and availability), indicating the inherent danger associated with it. The `security mechanism` perspective characterizes the specific security mechanism to which a vulnerability belongs, encompassing encryption of data, authentication processes, management of user sessions, authorization mechanisms, access restriction, input validation, and exposure limitation. The `attack surface` examines potential entry points for the attacks based on the vulnerability. This perspective includes the host system, the network, the running program or code.

6.1 Future Work

In advancing the field of software vulnerability classification, the resulting model of this work opens avenues for future work. One of them being the integration of the classifications provided by the developed model into existing entries within the CWE database. While some of these classifications are already present in the CWE database, others are missing. During mapping of the CWE entries to our model we have found that many classifications can be accomplished by semantic analysis of other fields of CWE entries (e.g. the extended description). Because doing this by hand consumes a lot of human resources and is error prone, an automated classification tool using semantic analysis and machine learning approaches could be a valuable further development to our work.

In addition this work establishes the foundation for the implementation of a comprehensive filtering of CWE entries based on all classifications provided by the model. This would allow for an easier and more intuitive access to CWE entries. The filtering could serve as a valuable tool to support developers and software architects in designing, implementing and maintaining secure software. For that, future work could explore how the model can be integrated and utilized in the whole cycle of secure software development.

7. REFERENCES

[1] M. Corporation. Cwe-303: Incorrect implementation of authentication algorithm, 2023. Accessed: 13.12.2023.

[2] M. Corporation. Cwe-601: Url redirection to untrusted site ('open redirect'), 2023. Accessed: 13.12.2023.

[3] M. Corporation. Cwe-79: Improper neutralization of input during web page generation ('cross-site scripting'), 2023. Accessed: 13.12.2023.

[4] M. Corporation. Cwe glossary, 2024. Accessed: 18.01.2024.

[5] A. Honkaranta, T. Leppänen, and A. Costin. Towards practical cybersecurity mapping of stride and cwe — a multi-perspective approach. In *2021 29th Conference of Open Innovations Association (FRUCT)*, pages 150–159, 2021.

[6] M. Humayun, M. Niazi, N. Z. Jhanjhi, M. Alshayeb, and S. Mahmood. Cyber security threats and vulnerabilities: A systematic mapping study. *Arabian Journal for Science and Engineering*, 45(4):3171–3189, 2020.

[7] P. K. Kudjo, S. A. Brown, and S. Mensah. Improving software vulnerability classification performance using normalized difference measures. *International Journal of System Assurance Engineering and Management*, 14(3):1010–1027, 2023.

[8] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws. *ACM Comput. Surv.*, 26(3):211–254, sep 1994.

[9] S. Moshtari, A. Okutan, and M. Mirakhorli. A grounded theory based approach to characterize software attack surfaces. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 13–24, 2022.

[10] N. I. of Standards and Technology. Vulnerability, 2023. Accessed: 10.12.2023.

[11] J. C. S. Santos, K. Tarrit, and M. Mirakhorli. A catalog of security architecture weaknesses. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 220–223, 2017.

[12] K. Tsipenyuk, B. Chess, and G. McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. In *Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics*, volume 500, page 36, 2006.

Comparison of Integration Testing Methods for Component-based Software

Maximilian Lucas
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
maximilian.lucas@rwth-aachen.de

Vincent Stollenwerk
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
vincent.stollenwerk@rwth-aachen.de

ABSTRACT

Integration testing is crucial in component-based software development and is essential to ensure the correctness and reliability of the system in today's complex software landscapes. Despite the existence of a large number of works on integration testing methods and the frequent emphasis on their importance, there is a lack of comprehensive comparisons of these methods. This paper bridges this gap by offering a classification of integration testing methodologies, highlighting their differences and main attributes in component-based software environments. Through a systematic literature review, we classified and analysed research papers and identified various characteristics in which integration testing methods differ, including the types of systems they target, the underlying testing approaches and implementation details, and the features they provide. With this paper, we aim to assist researchers in classifying integration testing methods, as well as software developers and testers, in comparing and selecting suitable integration testing methods.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering—*Software Testing*

Keywords

Software testing, Systematic Literature Review, Integration Testing, Component-based, Comparison

1. INTRODUCTION

In software development, ensuring correctness and reliability of software systems is of utmost importance. Myers et al. estimate that in a typical software project, 50% of time and more than 50% of costs are associated with the testing of the program [24]. Furthermore, studies indicate that the majority of errors in the software development life cycle are integration errors [3, 12] that would not be caught in unit tests. Therefore, integration testing is a crucial part of software engineering.

While there exists extensive literature on integration testing methods, there is a lack of comprehensive comparisons between integration testing methods. In conjunction with the heterogeneous nature of component-based software systems and integration testing methods, selecting the right approach and understanding their trade-offs can be complicated in practice. This paper aims to bridge this gap by comparing and analysing existing integration testing methods in a systematic literature review and offering a classification to simplify comparisons.

In the study, we have identified key characteristics of integration testing criteria for:

1. The kind of system the method targets
2. The underlying approach and implementation details
3. The features of the integration testing approach

First, we introduce the relevant background and definitions in Section 2. Then, we explain our research method in Section 3, document the review execution in Section 4, and develop and discuss our classification in Section 5. In the end, we discuss threats to the validity of our study in Section 6 and conclude our work in Section 7.

2. BACKGROUND

Before discussing our literature review, this section defines the relevant background and terminology. While we provide the IEEE definitions of our terminology to aid in understanding, during the literature review, we did not evaluate whether the proposed methods match our definitions and instead left the definitions to the authors of the articles.

Unit "1. separately testable element specified in the design of a computer software component 2. logically separable part of a computer program 3. software component that is not subdivided into other components [...]" [1]. In our context, a unit usually refers to one component of component-based software.

Unit test "1. testing of individual routines and modules by the developer or an independent tester 2. test of individual programs or modules in order to ensure that there are no analysis or programming errors [...] 3. test of individual hardware or software units or groups of related units" [1]. In our context, a unit test usually refers to a test that tests an individual component.

Integration test "1. progressive linking and testing of programs or modules in order to ensure their proper functioning in the complete system [...]" [1]. In our context, integration testing usually refers to tests that test the interaction between different components. Most integration testing techniques assume that the components function correctly individually, which can be ensured through unit tests.

Additionally, the paper contains some technical terms that are not necessary to understand the point of the paper, but might provide interesting context to the reader:

Context-sensitive middleware-based application Context-sensitive middleware-based applications are applications where a so-called middleware invokes different actions depending on the application state and the external context. [27]

An example could be an ice cream advertising software. If the weather (context) is cold and there is no actively running ad campaign (application state), the middleware does nothing. If then the weather gets warm (context change), the middleware could invoke the action to start the ad campaign.

Software product line Software product lines are a software engineering paradigm where multiple software systems are created from a shared core, which can be customised using variable modules. [26].

3. RESEARCH METHOD

Our Systematic Literature Review (SLR) was led by the guidelines published by Kitchenham [15]. In this section, we describe the review process and our research questions.

3.1 Research Questions

To provide a better way to compare integration testing methods, we have focused our SLR on three research questions:

RQ1 How can integration testing methods be classified according to the type of system to which they are applicable?

RQ2 How can integration testing methods be classified according to their implementation characteristics for testing?

RQ3 How can integration testing methods be classified according to the capabilities and features they offer in the software testing process?

We chose these research questions because we deemed the question of RQ1 – what system the approach targets – relevant both for researchers classifying approaches, as well as practitioners comparing and choosing a fitting approach. For RQ2, we believe that the implementation details are especially relevant for researchers who classify and develop integration testing methods, while RQ3 aims to assist software testers and engineers when selecting integration testing approaches.

3.2 Planning the Literature Review

In this section, we outline the methodology and strategic considerations for conducting the literature review. This includes our search procedures, the criteria for the selection of articles, and the data extraction strategy that we employed to ensure a comprehensive and systematic review of the relevant literature.

3.2.1 Search Procedure

When designing our approach to find and select articles, our goal was to obtain a broad overview of integration testing methods. Therefore, we kept our filtering and search process broad.

To find literature, we focused on the following sources:

- ACM Digital Library[1]
- Google Scholar[2]
- IEEE Xplore[3]
- Scopus[4]
- Springer Science Direct[5]

To keep our search queries broad, we only queried for "integration testing" and the terms "approach", or "technique". For Scopus we found that the results were better when we also included the term "framework". Because we wanted to focus on primary literature, we additionally excluded the terms "survey" and "literature review". Listings 1, 2, 3, 4, and 5 show the final queries we used for Google Scholar, Scopus, ACM Digital Library, IEEE Xplore, and Springer Science Direct, respectively.

```
allintitle: integration testing approach OR
↪  technique -"literature review" -survey
```

Listing 1: Google Scholar search query

```
TITLE ("integration testing") AND
↪  (TITLE("approach") OR TITLE ("technique") OR
↪  TITLE ("framework")) AND NOT
↪  (TITLE-ABS("survey") OR TITLE-ABS ("literature
↪  review")) AND (LIMIT-TO(SUBJAREA, "COMP"))
```

Listing 2: Scopus search query

```
[Title: integration testing] AND [[Title:
↪  approach] OR [Title: technique]] AND NOT
↪  [[Title: survey] OR [Title: "literature
↪  review"]]
```

Listing 3: ACM Digital Library search query

To maintain a manageable and relevant dataset for our review, we decided to utilise the first ten papers from Google Scholar and Scopus. This approach was chosen to ensure a balance between the comprehensiveness and manageability of the literature review. The top ten papers from each

[1]https://dl.acm.org/
[2]https://scholar.google.de/
[3]https://ieeexplore.ieee.org/
[4]https://www.scopus.com/
[5]https://www.sciencedirect.com/

```
(("Document Title": "Integration Testing") AND
↪  (("Document Title": technique) OR ("Document
↪  Title": approach)) NOT ("Document Title":
↪  survey) NOT ("Document Title": "literature
↪  review"))
```

Listing 4: IEEE Xplore search query

```
Title: "Integration Testing" AND (technique OR
↪  approach) -survey -"literature review"
```

Listing 5: Springer Science Direct search query

source were considered to be the most relevant and of the highest quality based on their ranking in the search results. This selection process allowed us to focus on highly relevant and high-quality studies, while also keeping the volume of literature to a practical level for in-depth analysis.

3.2.2 *Article Selection Procedure*

To ensure the relevance of the articles we include, we evaluated each search result against a list of inclusion criteria. First, we have identified and removed duplicates. In this paper, we consider articles to be duplicates if:

1. The previous search results, including those of other sources, already contain the same article
2. The article is a conference paper and the search results contain a corresponding journal article

After we filtered the results for duplicates, each remaining article was checked against the following inclusion criteria:

- The article must be open-access or licensed to the university library of the RWTH Aachen University
- The article must be a primary literature, not a literature review or survey
- The article must describe an integration testing method
- The article must be in the context of Software Engineering

3.2.3 *Data Extraction Strategy*

To ensure consistency and later answer our research questions, each paper was manually reviewed for three questions:

1. What kind of system does the method apply to?
2. What is the approach behind the method?
3. What are the features and attributes of the integration testing method?

Later, we used the answers to these questions to identify patterns and develop our classification.

4. REVIEW EXECUTION

Our queries resulted in a total of 1.560 results across all sources. Table 1 shows the number of results per source. As described in Section 3, to keep the dataset manageable, we only considered the first 10 results from Google Scholar and Scopus. A list of the results considered can be found in Table 2 for Google Scholar and Table 3 for Scopus. After removing duplicates, we were left with 19 articles. Out of these, we have removed 3 articles after filtering them with our inclusion criteria, leaving us with a list of 16 articles for review.

Data source	Number of results
Google Scholar	192
Scopus	39
ACM Digital Library	1309
IEEE Xplore	16
Springer Science Direct	4

Table 1: Results per source

To extract the relevant information and answer our research questions, we read each article and manually extracted the relevant data points. In the end, we have compared our findings and identified categories of targets, common implementation approaches, and common features.

5. RESULTS

This section outlines the results of our study. We present our findings following the order of the research questions: First, the classification of integration testing methods based on system applicability (RQ1), followed by their implementation characteristics (RQ2), and finally, their capabilities and features (RQ3).

5.1 Applicability

Generally, we have found three major groups of systems to which integration testing methods are targeted. In total, seven methods focus on object-oriented or aspect-oriented software. Three methods are technology-agnostic and focus on abstract component-based or modular systems, and another three focus on IoT-based systems. These three categories form the majority of target systems in our study. Additionally, we found three methods that target other, more specialised systems: Context-sensitive middleware-based applications, software product lines, and service-orientated architectures. A complete overview of which method applies to what kind of system can be found in Table 4.

5.2 Implementation Characteristics

In our research, we have identified a set of key characteristics and features that the methods provide or are based on. Each method can have multiple of these characteristics and can provide multiple features. Additionally, the list is not meant to be exhaustive. It is meant to provide a basis for high-level comparisons and could change as new approaches emerge.

We have identified the following list characteristics:

Architecture description-based (AD) The method uses a description of the architecture of the system as an input parameter.

In our study, the most common representation was UML diagrams. The approach foregrounds the generation and execution of integration test cases derived from architecture-specific models. Bertolino et al. introduce UML-based architectural models to generate integration test cases. This involves their so-called Computational Independent Test Model (CITM), Platform Independent Test Model (PITM), and Platform Specific Test Model (PSTM), which collectively de-

Ref. no.	Study Title	Included
[11]	Interface mutation: An approach for integration testing	✗
[2]	A state-based approach to integration testing based on UML models	✓
[25]	Integration testing in software product line engineering: a model-based technique	✓
[4]	An approach to integration testing based on architectural descriptions	✓
[16]	Integration testing object-oriented software systems: An experiment-driven research approach	✗
[9]	Integration testing of context-sensitive middleware-based applications: a metamorphic approach	✓
[14]	Development and application of a white box approach to integration testing	✓
[8]	A metamorphic approach to integration testing of context-sensitive middleware-based applications	✗
[7]	Contextual integration testing of object-oriented and aspect-oriented programs: a structural approach for Java and AspectJ	✓
[17]	Integration testing of object-oriented and aspect-oriented programs: A structural pairwise approach for java	✓

Table 2: First 10 search results in Google Scholar

Ref. no.	Study Title	Included
[23]	A Framework for Continuous Regression and Integration Testing in IoT Systems Based on Deep Learning and Search-Based Techniques	✓
[20]	An integration testing approach based on test patterns and MDA techniques	✓
[6]	PatrIoT: IoT Automated Interoperability and Integration Testing Framework	✓
[5]	Framework for Integration Testing of IoT Solutions	✓
[18]	An integration testing framework and evaluation metric for vulnerability mining methods	✗
[10]	Integration testing in the test template framework	✓
[21]	Towards Component-Based System integration testing framework	✓
[19]	A Unified test framework for continuous integration testing of SOA solutions	✓
[22]	A test framework for integration testing of object-oriented programs	✓
[13]	An effective model-based integration testing technique for component-based software	✓

Table 3: First 10 search results in Scopus

scribe the test objectives, test architecture, and platform-specific test artefacts. Class and sequence diagrams are extended using test patterns to formulate structured integration tests that align with the system's architecture and dynamic interactions. [4]

Control-flow graph-based (CF) The method involves a graph representing the control-flow of the software system. This can happen as an input or as an intermediary representation to work on.

In structural testing, a control-flow graph (CFG) is required to represent the structure of the program. The CFG is used to depict the flow of control within the program, where each node in the graph represents a statement or a block of statements that are executed sequentially, and each edge represents the flow of control transitioning from one statement or block to another [17].

State-graph-based (SG) The method involves a state-graph or state-machine describing the possible states of the software system and the existing transitions between them. This can happen as an input or as an intermediary representation to work on.

Mahmood, for example, uses a state-graph in their integration testing approach. To model component interactions, a Component Interaction Graph (CIG) is constructed based on collaboration diagrams from UML, which represent the interactions between components and show possible execution flows. A CIG consists of nodes representing components and directed edges representing control flow transfers between components, with each edge annotated by an interaction complexity metric. Through these directed graphs, interaction metrics are constructed, and weights are assigned to each interaction based on the frequency and content complexity of data types involved in exchanges between components. In the end, they used this graph to suggest tests based on certain coverage criteria such as described in the *configurable test criteria* section of Section 5.3. [21]

Machine-learning-based (ML) The method involves machine learning. In our study, use cases were the automatic discovery of the software architecture and functionality, as well as automatic test-case generation.

Specifically, Medhat et al. use machine learning to infer formal finite-state behavioural models of individual software components through an active learning approach. The method involves disassembling a complex integrated system into its constituent components, extracting approximated models as Mealy machines, and then constructing a product model to identify and test for compositional issues like deadlocks and live-locks. [23]

Unlimited integration depth (ID) The method does not limit the number of components that can be integrated and used in a single test case.

System under test	Ref. no.
Object-Oriented or Aspect-Oriented Software	[17], [7], [2], [20], [22], [21], [10]
Component-Based or Modular Systems	[14], [4], [13]
IoT-Based Systems	[23], [6], [5]
Context-Sensitive Middleware-Based Applications	[9]
Software Product Lines	[25]
Service-Oriented Architecture (SOA)	[19]

Table 4: Classification by system under test

Notably, the framework referenced in [17] is the only approach that restricts integration testing to pairwise integration tests. All other reviewed papers – while not always practical – specify no limit for integrated components exercised in a single integration test.

Specification-based (S) The method relies on formal specifications as the foundation for generating and executing integration test cases.

Cristia et al. introduce an approach that integrates model-based testing with formal specifications utilising the so-called Z notation. It is based on set theory and first-order predicate logic, which provides a precise and mathematically rigorous framework for modelling complex systems. Z notation is particularly well suited for describing system properties, developing design specifications, and verifying the correctness of system designs through mathematical proofs. [10]

Table 5 lists what criteria apply to which method. It is important to note that, since the proposed list is not exhaustive, fewer check marks do not necessarily mean fewer features.

Additionally, each characteristic comes with its trade-offs. Consequently, while our characterisation does highlight some key characteristics of the examined approaches, it can be used to compare methods, but cannot be used to judge the quality of the different methods.

Ref. no.	AD	CF	SG	ML	ID	S
[17]		✓				
[7]		✓			✓	
[14]		✓			✓	
[9]					✓	
[4]	✓		✓		✓	
[25]	✓	✓	✓		✓	
[2]	✓		✓		✓	
[23]				✓	✓	
[20]	✓				✓	
[6]	✓				✓	
[5]	✓				✓	
[10]	✓				✓	✓
[21]	✓	✓			✓	
[19]	✓				✓	
[22]	✓	✓			✓	
[13]			✓	✓	✓	

Table 5: Characteristics Results Summary

5.3 Capabilities and Features

In the reviewed papers, we have identified the following list of external-facing capabilities and features. Similar to the implementation characteristics described in Section 5.2, this list is not meant to be exhaustive and other features could be identified in the existing, or evolve with future methods.

Configurable test criteria (TC) The method provides multiple test criteria with different trade-offs.

In Section 5.2, we have identified that some integration testing approaches are graph-based. Many of these approaches suggest test cases by covering the generated graphs using different coverage criteria. Cafeo et al., for example, define their so-called $\mathcal{CoDU}$ graph, a graph that models the control flow of the system. They then define three testing criteria which define which elements of the program should be exercised. Two of these criteria require that either each node or each edge must be exercised by at least one test case, respectively [7]. Other graph-based approaches also include a path-coverage criteria where each path within some scope must be covered [21]. These coverage criteria provide trade-offs in the comprehensiveness of the tests compared to the required amount of test cases.

Approaches targeting IoT systems, on the other hand, often allow the user to configure what devices are exercised in the tests. Bures et al., for example, allow the users to configure testbeds that contain both physical and simulated devices. In their examples, simulated devices can be cheaper to test and simpler to handle, while bearing the risk of behaving differently than physical devices. [5, 6]

Automatic test generation (TG) The method can automatically generate test cases. Some approaches provide only guidance on the tests but cannot generate test cases automatically.

Maibaum et al., an architecture-based approach as discussed in Section 5.2, use UML diagrams to derive contracts between classes of object-oriented programs. These contracts are used to automatically generate Java code which exercises the components with test data that can be automatically generated or manually supplied. [22]

Other approaches, such as the graph-based approach proposed by Ali et al., cannot generate executable test cases but only generate a list of paths (test cases) that should be covered. [2] While the user is still prompted to manually generate the test data, this approach nevertheless lists all cases that the user should cover.

Approaches that do not meet this criterion, such as the approach proposed by Chan et al., only provide guidelines for designing test cases but no comprehensive list. [9]

Automatic test execution (TE) The method provides an environment or suite to automate the execution of tests.

Maciel et al. outline how automatic test case generation can be achieved using test patterns and Model-Driven-Architecture/Testing practices. Model-Driven-Testing is a technique that allows for the automated generation of test cases and the infrastructure needed to execute them on different platforms. They developed a tool to support this automatic approach, allowing the generation of test cases from models for object-oriented software systems. [20]

Liu et al. present a framework employing a flexible and extensible execution engine that can automatically execute test cases. This architecture is designed to understand test case behaviour logic and configuration files to schedule and execute tests. [19]

Bures et al. utilize a modified version of the JUnit testing framework, which has been enhanced to support automated interoperability and integration testing specific to Internet of Things (IoT) systems. The added features to JUnit include extra synchronization and orchestration capabilities, such as warning states and interruption points for automated tests, allowing for a more flexible management of the test step flow. [6]

Test case selection and prioritisation (SP) Aiming to maximize fault detection in large-scale systems, the method can prioritise executing certain test cases first to reduce the cost and effort of testing.

Mahmood et al., for example, employ complexity measures to identify and prioritise component interaction metrics and test adequacy criteria. [21]

Medhat et al. use deep learning and search-based techniques to prioritise test cases for IoT systems. [23]

Error handling and propagation (EH) These methods can examine how errors are handled within a component and how they are communicated to other components.

Cristia et al. explore the dependency-based order of component testing. The work suggests constructing accurate stubs or proving consistent interaction patterns between components to isolate and attribute errors to the correct source during integration tests. [10]

Table 6 summarises which set of features and capabilities apply to which method.

Ref. no.	TC	TG	TE	SP	EH
[17]	✓				
[7]	✓				✓
[14]					✓
[9]					✓
[4]					
[25]				✓	
[2]	✓	✓	✓		
[23]		✓		✓	
[20]		✓	✓		
[6]	✓		✓		
[5]	✓				
[10]	✓				✓
[21]	✓			✓	
[19]	✓	✓	✓		
[22]		✓	✓		
[13]	✓	✓	✓		

Table 6: Capabilities and Features Result Summary

6. DISCUSSION

In this section, we discuss the limits of our study and highlight potential threats to the validity of our study. Firstly, our final analysis was based on 16 papers. Although 16 papers might be enough to show fundamental characteristics and approaches, 16 papers cannot be considered representative of the whole research field. Therefore, while we believe our fundamental characteristics hold, we suspect that our lists are not exhaustive. Additionally, we were unable to draw conclusions about the distributions and trends among the different approaches.

Secondly, our initial search only included online sources and we only included articles from Google Scholar and Scopus. As a result, our study might exclude a significant body of work. Although we do not believe that this impacts our fundamental findings, a study including all online articles from more online and offline sources could lead to more exhaustive and more representative results.

Lastly, the data extraction process included manual reading and analysis of the articles by humans. While we have conducted the analysis strictly adhering to our predefined research method and our best abilities, this does leave room for human errors.

In conclusion, future studies that analyse a wider body of work and include more diverse sources might be necessary to find exhaustive results and draw representative conclusions. However, our foundational findings provide a strong basis for comparing integration testing approaches.

7. CONCLUSIONS AND FUTURE WORK

In this section, we provide directions for future research and conclude our paper. In this systematic literature review, we critically analysed existing integration testing methods, with a primary focus on component-based software systems. Our study identified key classifications based on system applicability, implementation characteristics, and the capabilities and features offered by various integration testing methods. The classifications provide a framework for understanding the diverse landscape of integration testing methods and comparing their suitability for different types of software systems, including object-oriented, aspect-oriented, IoT-based systems, and more.

7.1 Directions for Future Research

While our study laid a solid foundation for classifying and comparing integration testing methods, as discussed in Section 6, our list of characteristics is not exhaustive and the studies we have analysed are not representative of the broad field of integration testing. Therefore, a literature review that covers more articles and more sources, including other digital and physical libraries, could reveal more extensive findings. Similarly, a literature review that covers a larger and representative body of work and analyses the distribution of characteristics, as well as trends or correlations between the system under test and the different implementation characteristics and capabilities, might be interesting.

Additionally, in the methods we have analysed, we have identified several areas that raised questions and seem promising for further exploration:

- Machine Learning in Integration Testing: While we have analysed several promising approaches that use machine learning. In general, comparatively few approaches used machine learning. This leads us to believe that there is still untapped potential for the use of machine learning in integration testing. Especially automating test case generation and optimising test processes were barely explored in the articles we have covered. Research in this area could revolutionise the efficiency of integration testing.

- Performance and Scalability: As systems become larger and more complex, the performance and scalability of integration testing methods become critical. During our research, we found some methods that seemed impractical for real-world systems because they required a lot of computational work or test cases to be implemented. Future work should focus on optimising these aspects to handle large-scale systems efficiently.

- Human Factors in Integration Testing: Although the articles we analysed often covered extensive technical details, they usually did not discuss the applicability and developer experience of the proposed methods. Nevertheless, human factors such as cognitive load and developer experience are crucial for an effective software testing process, as in the end, methods need to be implemented and used by human engineers. Understanding the role of human factors in integration testing methods could lead to more user-friendly and efficient testing processes.

7.2 Conclusion

Our findings indicate that, while there is a wealth of research on integration testing methods, there remains a lack of consensus on the best approaches. The diversity in methodologies, from architecture description-based to machine learning-based approaches, underscores the complexity and evolving nature of software integration testing. We observed that most methods offer configurable test criteria and support automatic test generation, which is crucial for managing the increasing complexity of software systems.

While our systematic literature review provides a solid foundation for the current state of integration testing methods, it also highlights open questions and new developments in this field. We have identified different kinds of systems that are targeted by different approaches and highlighted the differences in their implementation and features. Additionally, for some implementation details and features, we have found relations, such as the use of different coverage criteria for graph-based approaches. Continuous research and adaptation are required to keep up with advancements in software development technologies and integration testing methodologies.

8. REFERENCES

[1] ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary. *ISO/IEC/IEEE 24765:2017(E)*, pages 1–541, Aug. 2017. Conference Name: ISO/IEC/IEEE 24765:2017(E).

[2] S. Ali, L. C. Briand, M. J.-u. Rehman, H. Asghar, M. Z. Z. Iqbal, and A. Nadeem. A state-based approach to integration testing based on UML models. *Information and Software Technology*, 49(11):1087–1106, Nov. 2007.

[3] V. R. Basili and B. T. Perricone. Software errors and complexity: an empirical investigation0. *Communications of the ACM*, 27(1):42–52, Jan. 1984.

[4] A. Bertolino, P. Inverardi, H. Muccini, and A. Rosetti. An approach to integration testing based on architectural descriptions. In *Proceedings. Third IEEE International Conference on Engineering of Complex Computer Systems (Cat. No.97TB100168)*, pages 77–84, Sept. 1997.

[5] M. Bures. Framework for Integration Testing of IoT Solutions. pages 1838–1839, 2018.

[6] M. Bures, B. Ahmed, V. Rechtberger, M. Klima, M. Trnka, M. Jaros, X. Bellekens, D. Almog, and P. Herout. PatrIoT: IoT Automated Interoperability and Integration Testing Framework. pages 454–459, 2021.

[7] B. B. P. Cafeo and P. C. Masiero. Contextual Integration Testing of Object-Oriented and Aspect-Oriented Programs: A Structural Approach for Java and AspectJ. In *2011 25th Brazilian Symposium on Software Engineering*, pages 214–223, Sept. 2011.

[8] W. Chan, T. Chen, H. Lu, T. Tse, and S. Yau. A metamorphic approach to integration testing of context-sensitive middleware-based applications. In *Fifth International Conference on Quality Software (QSIC'05)*, pages 241–249, Sept. 2005.

[9] W. K. Chan, T. Y. Chen, H. Lu, T. H. Tse, and S. S. Yau. Integration testing of context-sensitive middleware-based applications: a metamorphic approach. *International Journal of Software Engineering and Knowledge Engineering*, 16(05):677–703, Oct. 2006. Publisher: World Scientific Publishing Co.

[10] M. Cristiá, J. Mesuro, and C. Frydman. Integration testing in the test template framework. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8411 LNCS:400–414, 2014.

[11] M. Delamaro, J. Maidonado, and A. Mathur. Interface Mutation: an approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, Mar. 2001. Conference Name: IEEE Transactions on Software Engineering.

[12] S. Eldh, S. Punnekkat, H. Hansson, and P. Jönsson. Component Testing Is Not Enough - A Study of Software Faults in Telecom Middleware. In A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, editors, *Testing of Software and Communicating Systems*, Lecture Notes in Computer Science, pages 74–89, Berlin, Heidelberg, 2007. Springer.

[13] A. Elsafi, D. Jawawi, A. Abdelmaboud, A. Ibrahim, and I. Almahy. An effective model-based integration testing technique for component-based software. 2019.

[14] A. Haley and S. Zweben. Development and application of a white box approach to integration testing. *Journal of Systems and Software*, 4(4):309–315, Nov. 1984.

[15] B. Kitchenham and S. Charters. Guidelines for performing Systematic Literature Reviews in Software Engineering. 2, Jan. 2007.

[16] Y. Labiche. Integration testing object-oriented software systems: An experiment-driven research approach. In *2011 24th Canadian Conference on Electrical and Computer Engineering(CCECE)*, pages 652–655, May 2011.

[17] O. A. L. Lemos, I. G. Franchin, and P. C. Masiero. Integration testing of Object-Oriented and Aspect-Oriented programs: A structural pairwise approach for Java. *Science of Computer Programming*, 74(10):861–878, Aug. 2009.

[18] J. Li, J. Chen, M. Huang, M. Zhou, W. Xie, Z. Zeng, S. Chen, and Z. Zhang. An integration testing framework and evaluation metric for vulnerability mining methods. *China Communications*, 15(2):190–208, 2018.

[19] H. Liu, Z. Li, J. Zhu, H. Tan, and H. Huang. A Unified test framework for continuous integration testing of SOA solutions. pages 880–887, 2009.

[20] C. Maciel, P. Machado, and F. Ramalho. An integration testing approach based on test patterns and MDA techniques. volume 23-27-September-2010, 2010.

[21] S. Mahmood. Towards Component-Based System integration testing framework. volume 2, pages 1231–1235, 2011.

[22] T. Maibaum and Z. Li. A test framework for integration testing of object-oriented programs. pages 252–255, 2007.

[23] N. Medhat, S. Moussa, N. Badr, and M. Tolba. A Framework for Continuous Regression and Integration Testing in IoT Systems Based on Deep Learning and Search-Based Techniques. *IEEE Access*, 8:215716–215726, 2020.

[24] G. J. Myers, C. Sandler, and T. Badgett. Front Matter. In *The Art of Software Testing*, pages i–xi. John Wiley & Sons, Ltd, 2012.

[25] S. Reis, A. Metzger, and K. Pohl. Integration Testing in Software Product Line Engineering: A Model-Based Technique. In M. B. Dwyer and A. Lopes, editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 321–335, Berlin, Heidelberg, 2007. Springer.

[26] V. Sugumaran. Software Product Line Engineering. Jan. 2006.

[27] T. Tse and S. Yau. Testing context-sensitive middleware-based software applications. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, pages 458–466 vol.1, Sept. 2004.

A Survey of Jupyter Notebook Quality Attributes

Leon Carmincke
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
leon.carmincke@rwth-aachen.de

Lukas Jansen
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
lukas.maximilian.jansen@rwth-aachen.de

ABSTRACT

Jupyter Notebooks are an interactive, web-based tool widely adopted in various domains, allowing users to share computational documents in over 40 programming languages. Defining and assessing quality attributes is crucial for creating and using Notebooks to ensure reproducibility and overall effectiveness. Current studies of public Notebooks highlight the challenges and opportunities, such as ensuring code quality and enhancing understandability. However, there is no overview of the quality attributes and their current state. They are either mentioned implicitly, or the various perspectives haven't been integrated to provide researchers with a clear map of the current landscape of quality attributes.

Using a systematic mapping study, we review the current state of research. We selected 22 papers explicitly or implicitly mentioning quality attributes and identified reproducibility, understandability, and code quality as the leading attributes. In addition, we found limited consensus in attaining and evaluating quality standards. However, there is a common alignment on the quality attributes themselves. In general, the actual quality of publicly available Notebooks falls short of the respective standards regardless of how they are defined. Based on our overview, future work could suggest solutions like a quality model or tools based on the attributes.

Categories and Subject Descriptors

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

Keywords

Jupyter Notebooks, Quality Attributes, Literature Survey

1. INTRODUCTION

Jupyter Notebooks have become an essential tool in software projects and scientific research. They change how ideas are prototyped, experimented with, and shared by integrating code, visualizations, and textual explanations all in one web-based document. The literate programming paradigm, introduced by Donald Knuth in 1984, describes exactly this approach: combining code and natural language to document the logic of a program and thus improving understandability [17, 12, 19]. Consequently, it removes barriers to tracing experiments, ideas, and results, especially for more technical novices engaging in coding and data science. This facilitates reproducibility by providing a transparent and executable narrative.

The Notebooks are structured in so-called cells, which are blocks that contain either code or Markdown text [1]. Code cells can be written in over 40 programming languages. Some of the most popular are Python, R, and Julia, while 93.11% of the Notebooks use Python [17]. Markdown cells are used for explanations, documentation, and other content. Each cell can be run independently. While code cells are executed, Markdown cells are rendered. This also means that the cells do not have to be executed linearly. The results of code cells, if any, are written in new cells below, while graphics are directly rendered.

The interactive documents close the gap between computation and explanation, leading to increased use of Notebooks in various fields such as data science, machine learning, academic research, and industrial applications [26, 8, 4, 20, 23, 28]. Some fields include Notebooks as interactive documentation and examples for software projects or prototypes for project acquisition. Also, Notebooks are the tool of choice for data scientists, enabling researchers to document their experiments and publish their results [5, 31] in a new, directly executable form. Furthermore, the popularity of Notebooks in education has been steadily growing over the past few years [2, 23]. However, with the increased usage, quality has become a critical issue for effective collaboration, reproducibility, and knowledge transfer. Common issues include unexpected behavior due to out-of-order execution, poor coding practices that hinder readability and maintainability, and dependency problems that make Notebooks hard to run in different environments [4, 16]. This excerpt of issues influences the successful use and distribution of Notebooks in various fields. Best practices and a definition of quality need to be established to enable disciplined and informed usage.

This survey outlines key quality dimensions for a Notebook, ranging from code to storytelling. We systematically examine the landscape of quality attributes from a broad overview to a detailed definition and analysis of the current

SWC Seminar 2023/24 RWTH Aachen University, Germany.

state and their interrelationships. This approach improves the understanding of the various factors influencing the overall quality of Notebooks.

Our paper has the following structure. First, Section 2 overviews related papers that give a broad overview of different quality aspects to discuss their insights and limitations. Section 3 describes the methodology employed and the resulting papers in general and for each step during the process. Our results are mapped to seven quality attributes and described in Section 4. Chapter 5 summarizes the quality attributes and gives an outlook on further improvements and possible developments.

2. RELATED WORK

Since the spin-off of Jupyter Notebooks from its predecessor project, iPython, in 2014, it has become widely popular among researchers, data scientists, and professionals in various fields, making it a key tool for collaborative and data-centric work [17, 28, 20].

Research has diversified, focusing on improving transparency, highlighting collaborative practices, and establishing standards and best practices to meet the evolving needs of users in different areas. This evolution in research reflects the nature of Notebooks and their increasing significance. Numerous papers discuss the quality attributes from various perspectives and propose methods and tools to improve them. This section presents three papers broadly studying Notebook quality attributes from different domains, serving as exemplary cases for the approach in the literature.

Pimentel et al. conducted a large-scale study of 1.4 million public Notebooks from GitHub and analyzed their characteristics that impact quality and reproducibility [17]. The study shows that out of over 860,000 Notebooks with defined Python versions, dependencies, and execution order, only 4.03% produce the same results when re-executed, thus indicating low reproducibility. They proposed the tool Julynter, a Jupyter Lab extension that identifies and suggests modifications to improve Notebook reproducibility. The paper provides a rich insight into the current state of the quality of public Notebooks, especially reproducibility, and is thus one of the building blocks of our analysis. In this extensive work, many attributes that are discussed in the literature have been elaborated upon. Our study uses several such publications from various fields to define and discuss the current state of the quality attributes. This also includes proposed tools to support the realization, such as Julynter, to summarize the existing landscape for further research, and Notebook developers as well.

Candela, Chambers, and Sherratt assessed the quality of Jupyter projects published by Galleries, Libraries, Archives, and Museums (GLAM) organizations [3]. Their analysis is based on seven dimensions: understandability, availability, efficiency, traceability, portability, recoverability, and credibility. Different metrics were defined for each of these dimensions to describe the realization. In contrast to the previous study, the analysis of the GLAM domain is limited to 11 Notebook projects. However, it underlines that understandability is achieved through a good ratio of Markdown to code cells with descriptive names. It is also shown that the analyzed Notebooks do not contain a date when they were last run, which could negatively impact the reproducibility in this field. The study does not analyze the code in the Notebooks regarding software engineering best practices. The focus on GLAM organizations adds a unique perspective, although a deeper analysis of coding practices would be beneficial.

Other papers include clear guidelines for creators of Notebooks to improve quality, such as the ten rules for writing and sharing computational analyses in Notebooks published by Rule et al. [21] or the paper ”Best Practices for Collaboration with Computational Notebooks“ by Quaranta et al. [19] These rules range from the storytelling level over the code to the sharing experience. Both papers consistently highlight the documentation and structure of the Notebook. Additionally, we find information here about the version management of Notebooks and a process description for abstracting Notebooks for various use cases. However, the first paper does not fully reflect the current literature, as the rules were gathered during a workshop, and the impressions collected are summarized. Despite this limitation, the guidelines offer valuable insight to enhance the quality and communicative effectiveness.

We contribute to the literature regarding the quality of Notebooks by giving an extensive overview of quality attributes and their current state. The presented exemplary papers form the foundation for our work. We combine the insights from different publications in our systematic mapping study to help researchers navigate the field of quality attributes and build upon them in future work.

3. METHODOLOGY

As this study aims to overview the existing research, we conducted a systematic mapping study. First, we formulated our research questions:

RQ1: What are the currently researched quality attributes of Jupyter Notebooks?
First, we must find a general overview and definition of all the quality attributes in current research. Therefore, we start our study with this general question.

RQ2: What is the current state for the identified quality attributes in Notebooks?
Building upon the identified attributes, we discuss the current state of evaluation and assessment methodologies for them and to what extent these attributes are respected in publicly available Notebooks.

RQ3: How are the quality attributes related to each other?
After answering both questions, we analyze the found attributes regarding their relationship.

3.1 Search Process

At first, we had to decide which database to use for the search. Because the process is fairly complex and we wanted to reduce the additional overhead of using multiple databases, we decided to use a single database, Scopus. We deem this acceptable as the content of the other two large databases in software engineering, ACM Digital Library and IEEE Xplore, are also indexed by Scopus. Additionally, a single missing paper will not influence our results because we aim to give a general overview. Scopus also enables us to easily create complex queries and perform snowballing by searching for citing papers and references.

The general idea for our search, visualized in Figure 1, is as follows: Start with an initial query, seen in Figure 2, that will result in a small amount of highly relevant papers. From there on, we manually filtered out papers by using these exclusion criteria:

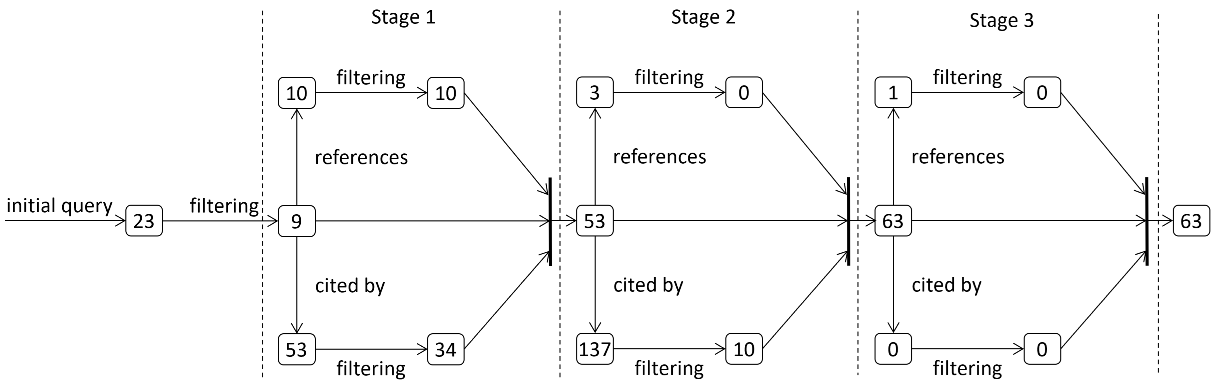

Figure 1: Overview of the Search Procedure

- **Paper does not have a DOI.** This criterion is critical for our search process as we use the DOI as the unique identifier for each found paper. This allows us to keep track of papers that we excluded. Additionally, if a paper does not have a DOI, it hints at a low quality. This also excludes materials without DOIs, like books, from our results.
- **Notebooks are not the focus of the paper.** Many papers use Notebooks to accomplish tasks, e.g., training an ML model or publishing results. However, the Notebook itself is often not the focus in this context.
- **The quality of Notebooks is not covered.**

```
TITLE-ABS-KEY (jupyter AND quality)
AND ( KEY (jupyter) OR TITLE (jupyter) )
AND ( SUBJAREA (COMP) )
AND ( LIMIT-TO (LANGUAGE , "English") )
```

Figure 2: Query Used for the Initial Search

The initial query about the ambiguous papers is acceptable in this new research area, as few papers exist.

After filtering out the irrelevant papers, we started the snowballing process, in which we looked at the references of the found papers and at papers that cited the found papers. We added a relaxed version of the initial query, seen in Figure 3 as an automatic filter for the snowballing results. This reduced the results from multiple hundreds down to manageable amounts. We also applied the same manual filtering to these additional papers. We conducted multiple iterations of this process, called stages, in Figure 1. The next iteration snowballs on all the papers found up to that point. These iterations naturally conclude because, at a certain point, no additional papers will be added to a stage; thus, there is nothing to snowball. This was the case after *Stage 3*. At that point, we had accumulated 63 papers.

```
SUBJAREA(COMP) AND TITLE-ABS-KEY(jupyter)
```

Figure 3: Query Used during the Snowballing

We were conservative in excluding papers, especially in *Stage 1*, because even slightly relevant papers may lead to a more relevant paper in the next stage.

3.2 Search Results

Based on Figure 4, the growing interest in Notebooks over time can be seen.

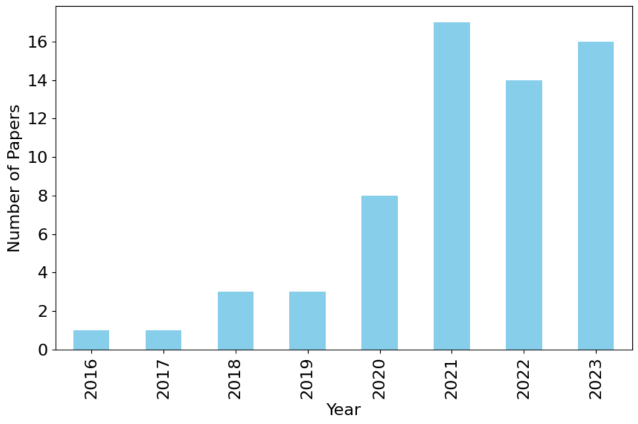

Figure 4: Publication Years of the Resulting Papers

Based on the search process, not every paper was relevant to answering our first research question. Therefore, we manually looked at the full text of the papers to find the ones suitable for extracting the results. During this search, we also came up with the seven attributes we analyzed in the results. These either came up directly in the papers or were introduced indirectly. To get an overview of the papers, we looked at how many papers mentioned the attributes or synonyms, e.g., ”testability“, ”testable“, or ”testing“. There were 24 papers, including code quality, 11 including understandability, 39 including testability, 58 including reproducibility, 23 including reusability, 33 including usability, and 50 including shareability. From these papers, we chose 22 papers as the basis of our analysis.

4. RESULTS

Analyzing the quality of Jupyter Notebooks involves examining different levels, aligning with well-established software quality attributes. Code cells responsible for functionality represent one facet, while the description level measures how well the individual cells are documented and explained. Given the diverse audience, effective communication and comprehensive descriptions are key to facilitating collaboration between people from different backgrounds [25].

We have extracted various quality attributes from the literature that can be categorized into three groups based on their relationships with each other and characteristics: development, storytelling, and collaboration. The following sections delve into a detailed description of each quality attribute, accompanied by a precise definition, answering RQ1 and RQ2. Furthermore, the current state of each attribute will be discussed, illustrating their relationships and classification into one of the three categories to answer RQ3.

Given that over 90% of Notebooks are written in Python, this analysis focuses primarily on this programming language. We limit the following analysis of the quality attributes *code quality* 4.1, *reusability* 4.2, and *testability* 4.3 to this programming language.

Table 1: References Utilized in the Analysis of Each Attribute

Categorization	Attribute	References
Development	Code Quality	[6, 25, 33, 18, 5] [7, 17]
	Reusability	[13, 21, 25, 17, 11] [22]
	Testability	[17, 4, 19]
Storytelling	Understandability	[17, 5, 4, 12, 21]
Collaboration	Reproducibility	[17, 34, 31, 15, 27]
	Shareability	[3, 17, 21, 24]
	Usability	[32, 9]

4.1 Code Quality

Code Quality is one of the critical quality aspects in software engineering [6]. However, in Jupyter Notebooks, the code is not always the primary output. Notebooks are used, among other things, for exploratory data analysis and scientific computing, where the result or insight often takes precedence over the code [25]. On the other hand, valuable scientific results need to be consistently verifiable and reproducible, which is directly affected by the code quality. Moreover, areas such as teaching at university rely on best practices in code quality [23] to create a role model for the students. The transition from Notebooks used as prototypes into production software also benefits from high quality and a software engineering-aware approach during the development of the Notebook [18]. Overall, high code quality facilitates long-term collaboration and further extensibility.

Definition.

Code quality is the degree to which code conforms to standards and best practices that ensure its readability, maintainability, reliability, and efficiency. Regarding Notebooks, we focus on the programming language Python, so good quality must respect the Python coding style [30], and maintainability standards [33, 14].

Features that define code quality are cyclomatic complexity, cognitive complexity, duplication, code smells, technical debt, maintainability, and reliability [5]. Proper variable naming and code structure are mandatory for understandability, including the absence of standard code smells, such as missing whitespace or excessively long code lines. Due to the cell layout, the structure influences the overall code quality [29]. Third-party libraries influence code quality due to potential compatibility issues, maintenance challenges, and the impact on code readability. Additionally, relying on deprecated functions can compromise code quality by introducing security vulnerabilities and hindering the adoption of modern coding practices [33].

Current State.

Studies have shown that public Notebooks suffer from poor code quality [33]. Writing code directly in Notebooks is prone to errors and bad software engineering principles as the support by the editor is limited compared to a full Integrated Development Environment (IDE) [7]. There is a lack of available resources, including proper version control, to improve code quality, resulting in poor programming habits. Additionally, the analysis by Wang et al. of the usage of the popular Python library SCIKIT-LEARN shows that around 35.55% of Notebooks use deprecated APIs [33].

Recent tools such as PYNBLINT [18] tackle this issue by providing linting functionalities, i.e., checking the code against a predefined set of rules. This general static analyzer can be used on all Notebooks, regardless of the platform used to write them.

Relation to Other Attributes.

Code quality is foundational to several other quality attributes and is assigned to the development category. It directly impacts *understandability* 4.4, *reusability* 4.2, and *testability* 4.3.

4.2 Reusability

Our analysis considers the reusability quality attribute from a functional perspective to examine which parts of a Jupyter Notebook can be functionally reused. Therefore, reusability is, in a sense, an extension of code quality with a focus on modularization. Promoting reusability is a key factor in preventing both code duplication and fragmentation.

However, in the case of Notebooks, reusability interferes with reproducibility, as it requires in complex cases to decouple code from the Notebook in external modules. This approach can lead to issues since the Notebook is no longer self-contained, meaning not all data necessary to reproduce the result is in one file [13].

Definition.

Reusability refers to how parts of a Notebook can be reused in other projects and applied in different contexts and purposes. Reusability can be achieved by creating modules, libraries, or functions that can be imported or called from other Notebooks. Reusability can also be facilitated by tools that allow users to search, browse, and insert snippets from other sources [13].

Reusability can also be considered as the possibility of reusing the whole Notebook. If the structure of a Notebook allows for easily changing out the input data, then the whole Notebook can be reused [21]. However, this approach naturally leads to code duplication.

Current State.

Modularity is the standard approach when developing large-scale software. However, Notebooks have no standardized approach to, for example, including the contents of one Notebook in others [25]. Few Notebooks use parameters or configuration files to customize their execution [17]. It is possible to extract functions to own Python packages when they are general enough to be applied to different contexts and purposes or when they are too complex or lengthy to be included in the Notebook. Most users do not extract code into modules, as only around 10% of Notebooks use local imports [13]. This hinders the reusability and testability of Notebooks [17].

A study of 2.5 million public Notebooks on GitHub shows that 50% have no unique code cell, meaning they only consist of copied code from other Notebooks [11]. The authors of [11] state that instructions from tutorials or course material also possibly cause such a high number.

An approach to encourage the reusability of Notebooks is cell folding [22]. This allows users to hide and show groups of cells. The new organization and annotation of cells enable different forms of reuse, as the parts that are not relevant for further work can just be collapsed.

Relation to Other Attributes.

Reproducibility 4.5 is a prerequisite for *reusability.* In Notebooks, this attribute is also related to *understandability* 4.4, i.e., how well the content is documented. The reusability attribute is categorized into the "Development" category due to the high connection to *code quality.* It might also be categorized into the "Collaboration" category because reusability improves collaboration, as others can share and build on existing functionality.

4.3 Testability

Testing code is an elementary aspect of software development that ensures the quality and functionality of the code. By writing tests, developers verify that their code meets its requirements and functions as intended, also after changes are applied [10].

Testability, similar to code quality, is often not a top priority in Notebooks. Data scientists, among others, frequently use Notebooks for fast and rough exploration and analysis. As a result, they may not give high priority to testing or following guidelines for code quality and documentation [4].

Definition.

The testability of a Notebook refers to how easily its functionality can be evaluated through testing. A well-organized and clean notebook design favors effective testing for coverage of all functionality. Improving testability is done by creating functions for complex code and defining a baseline for expected test results [17].

Current State.

There is no universally accepted methodology for testing code in Notebooks [4]. Tests can be directly embedded within the Notebook, but challenges exist in establishing a standardized testing approach. An analysis of a large dataset of public Notebooks from 2019 shows that only 1.54% import known Python test modules [17]. However, a minority of repositories test code associated with the Notebook outside the Notebook environment.

This highlights the opportunity to improve testing practices, especially for features with more complex code. A suitable test suite ensures the code can be reproduced in different environments. For Notebook code focusing on data exploration, studies have shown that current tools are unsuitable and can be too complex [17]. Additionally, necessary integrations are not available that let users run tests and show the results and metrics such as test coverage [19].

Relation to Other Attributes.

Testability is crucial for the reliability of Notebooks. It is linked to *code quality* 4.1, as well-tested code contributes to higher overall *code quality.* It also affects the *understandability* 4.4 and *reproducibility* 4.5 of Notebooks.

4.4 Understandability

Literate programming describes writing code along with natural text [12]. From a purely software-engineering point of view, this concept has met with criticism, as good code should be self-explanatory, and detailed documentation is time-consuming to maintain.

However, other areas benefit from such an approach. Domains such as research, natural science, or data science want to convey complex analyses or results with often theories or equations behind them [12]. The benefits of enriching the source code with various forms of documentation side-by-side are evident. This leads to the conclusion that the quality of a Notebook needs to be observed from at least two perspectives: software engineering and storytelling.

Definition.

Understandability refers to the ease with which a user can comprehend the content. This contains various metrics and attributes that contribute to the overall clarity of the Notebook. One elementary factor influencing understandability is the ratio of Markdown to code cells, as highlighted by Pimentel et al. in their work on quality and reproducibility [17]. The textual descriptions can also be measured with the Flesch readability score [5]. In addition, the naming of the Notebook should be distinctive, and generic terms such as "Untitled" should be avoided to ensure that it immediately conveys the purpose or content of the Notebook.

Current State.

Starting with the name of the Notebook file, around 1.99% of public Notebooks are named "Untitled", and 0.69% have "Copy" in their name [17]. Many public Notebooks have insufficient descriptive cells so that users can understand all of their content [4, 12, 21]. Developers improve the understandability by rearranging code cells rather than extensive use of explanatory Markdown [12]. A minimum of twice as many Markdown cells than code cells indicates the literate programming paradigm [32, 31] and therefore good documentation. In any case, extensive documentation at the start and end of the Notebook can improve the understandability significantly [17].

Relation to Other Attributes.

Understandability is linked to *code quality* 4.1 and *reproducibility* 4.5. Since it is heavily influenced by the literate programming paradigm, it falls under the category of "Storytelling".

4.5 Reproducibility

Reproducibility in Jupyter Notebooks involves the ability to recreate and validate results by executing code cells [21]. On the one hand, reproducibility is closely tied to understandability, i.e., how well the code is documented. On the other hand, the execution order of the code cells, dependency management, and data availability impact reproducibility. This includes the presence of a baseline, which means that all code cells have outputs from a successful execution. This way, users can verify the outputs after running the code locally. In this Section, we focus on the reproducibility on their own, independent of, for example, the underlying code quality.

Definition.

A Notebook is reproducible if it can be executed without errors and its outputs match the original ones. This includes the ability of a user to trace the original execution sequence to produce a given result.

Current State.

Studies have shown that many Notebooks are not reproducible [17, 34] due to randomness, time, and data or a missing definition of the execution environment [24]. Pimentel et al. show that many Notebooks with unambiguous execution sequences have unexecuted code cells, out-of-order cells, and execution jumps in the execution counter, making it difficult to track the execution status [17]. However, it is indicated that excluding execution jumps at the beginning reduces their frequency, possibly indicating that re-execution without a Notebook restart causes these jumps.

Additionally, sufficient markdown cells increase navigability and understandability, improving reproducibility. Currently, Notebooks suffer from poor documentation, as over 30% of public Notebooks do not contain a single markdown cell [17] and as developers tend to neglect the literate programming paradigm when writing code [15].

The two tools HeaderGen and Cell2Doc [31, 15] aim to automatically document the provided code in Notebooks in the data science field. HeaderGen focuses on machine learning Notebooks by carrying out a static code analysis of the code cells and libraries to automatically categorize functions regarding their purpose in a typical machine learning pipeline. These headers, one to three words long, provide a basic navigation structure and information regarding the structure and purpose of each section in data science notebooks. Furthermore, there exist tools such as the Osiris prototype [34] that can achieve a reproducibility rate of 82.23%, more than three times better than the state-of-the-art. This tool reconstructs possible execution orders based on the dependencies between cells.

Histree [27] visualizes the execution order of code cells as a tree to combat bad reproducibility due to the execution order of cells. This also allows researchers to trace and reproduce different experiment sequences representing results in a natural form.

Relation to Other Attributes.

Reproducibility is closely tied to *code quality* 4.1, *testability* 4.3, and *understandability* 4.4. Well-documented code and comprehensive testing contribute to the reproducibility of results.

4.6 Shareability

Shareability focuses on the collaborative aspects of Jupyter Notebooks. The main objective is to evaluate the features that allow seamless sharing, such as clear instructions, the accessibility of dependent data, and a list of libraries used.

Definition.

Shareability defines the process of allowing foreign users to work with the Notebook. It depends on the Notebook's license and other factors such as the size [3]. The execution environment must be documented containing the Python version and dependency management [17, 24].

Current State.

Popular Notebooks tend to have higher quality and are easier to share and reproduce than the overall corpus, but there is no clear correlation between quality and citation [17]. To provide access to Notebooks, the depending data and software must be available for the user, which can be difficult if the project size increases [21]. Pimentel et al. examined that over 79.58% of repositories with dependencies use a requirements.txt file to document the Python requirements [17]. Creating the same environment where the Notebook should be executed on every machine is important, improving shareability and reproducibility, as the correct version is pinned in the requirements file.

Relation to Other Attributes.

Shareability is related to the *reproducibility* 4.5 and *understandability* 4.4.

4.7 Usability

The usability of a Notebook depends on several quality attributes. The quality of the Notebook's components, such as text, multimedia, and code is crucial for its usability. It is important to consider the readability, whether it flows cohesively like a book or appears as a collection of disjointed code cells. Moreover, it is essential to examine the Notebook's interactivity, how results are presented, and the potential for user interaction, especially in visual elements like charts [32].

Definition.

Usability defines the overall user experience, including text presentation, multimedia elements, and code. It's about whether the Notebook follows a coherent structure that makes reading, understanding, and interacting with the content easy.

Current State.

Design features enhance the usability of Notebooks. These features include a combination of HTML and Markdown elements, a navigation pane, alert boxes highlighting prerequisites, and anchor links allowing easy navigation within the Notebooks [32]. The use of backticks in Markdown cells emphasizes important sections or code. These features make the Notebooks adaptable to different modalities, such as educational content, for example, serving as effective materials for both instructor-led courses and self-paced learning. However, these aspects are less discussed in current literature, which focuses more on functionality and less on ultimate usability. Only well-documented and expressive Notebooks are suitable for people new to a domain, and well-thought design choices contribute positively to usability [32].

Providing a Notebook via services like Binder lowers the entry barrier drastically [9]. The user only requires a web browser to use a specific Notebook, as the environment is created on-demand in the cloud. This is handy for all short-lived interactions where users want to quickly experience and read the content without the hassle of creating the local installation and preparing the environment. This includes educational content and, for example, documentation of software libraries that benefit from interactive examples.

Relation to Other Attributes.

Usability is a natural extension of *reproducibility* 4.5 and *understandability* 4.4. A well-structured and readable Notebook enhances the user experience, contributing to the overall usability.

4.8 Summary

Our literature review has identified seven quality attributes divided into three categories: development, storytelling, and collaboration. The development of Notebooks includes the following three attributes: code quality, reusability, and testability. The understandability attribute represents the storytelling level, while reproducibility, usability, and shareability fall into the collaboration level.

These levels correspond to different stages of the Notebook lifecycle. The development level focuses on coding and software engineering practices. The understanding level emphasizes how well others can comprehend the content, while the collaboration level addresses attributes that promote effective collaboration and knowledge sharing.

The attributes most frequently mentioned in the literature are *code quality*, *understandability*, and *reproducibility*. When attributes mutually influence each other, the impact of one attribute on the other is mostly positive, with the only exceptions being *reusability* and *reproducibility*. We observed that increased code reusability is associated with modularization, meaning code is outsourced from the Notebook, which can negatively impact the reproducibility of a Notebook since the entire code is no longer in one place. Each project must decide this tradeoff, which often depends on its complexity. Due to the diverse applications of Notebooks and the resulting varied environments, our analysis does not claim to be complete. For instance, we limit ourselves to the most popular programming language, Python, although Notebooks support many other programming languages.

5. CONCLUSION AND FUTURE WORK

The results show that the quality of Jupyter Notebooks depends on multiple factors that are each unevenly adapted in today's Notebooks. We depict a gap between the desired quality and the current state-of-the-art. However, there is promising research that describes these problematic areas and proposes solutions. High-quality Notebooks utilize the literate programming paradigm to create a narrative throughout the content. This is essential for understandability and, consequently, reproducibility. Additionally, the code is a central element, currently often containing clones and difficult to test, but directly executable and embedded in the description. Collaborating through Notebooks is a further pillar. This contains usability, which becomes especially important when novices or beginners are the target audience for educational content.

Overall, our study describes the current landscape of quality attributes utilizing extensive research from different fields. This guides researchers in improving the quality of Notebooks as we discuss the current state documented in several studies as well as approaches and tools to improve quality.

Future Work.

Building on the presented systematic literature review, future work could involve developing a quality model for Notebooks similar to ISO/IEC 25000. This model could be a structured framework for evaluating and improving various facets of Jupyter Notebook quality. Additionally, tools such as JULYNTER [17] or HISTREE [27], built around quality attributes, present an opportunity to introduce and analyze them and their potential impact on the quality of Notebooks. Understanding how these tools align with existing practices can help close identified quality gaps. This is an essential contribution to establishing best practices of Notebook development and use.

6. REFERENCES

[1] Project Jupyter Documentation. `https://docs.jupyter.org/en/latest/`, Accessed November 19, 2023.

[2] Computational and Mathematical Modeling Program. `https://www.cammp.online/`, Accessed October 7, 2023.

[3] G. Candela, S. Chambers, and T. Sherratt. An Approach to Assess the Quality of Jupyter Projects Published by GLAM Institutions. *Journal of the Association for Information Science and Technology*, 2023.

[4] S. Chattopadhyay, I. Prasad, A. Z. Henley, A. Sarma, and T. Barik. What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities. In R. Bernhaupt, editor, *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, ACM Digital Library, pages 1–12, New York, USA, 2020. Association for Computing Machinery.

[5] M. Choetkiertikul, A. Hoonlor, C. Ragkhitwetsagul, S. Pongpaichet, T. Sunetnanta, T. Settewong, V. Jiravatvanich, and U. Kaewpichai. Mining the Characteristics of Jupyter Notebooks in Data Science Projects, 2023.

[6] B. Curtis. Measurement and Experimentation in Software Engineering. *Proceedings of the IEEE*, 68(9):1144–1157, 1980.

[7] T. L. de Santana, P. A. M. S. Da Neto, E. S. de Almeida, and I. Ahmed. Bug Analysis in Jupyter Notebook Projects: An Empirical Study, 2022.

[8] H. Dong, S. Zhou, J. L. Guo, and C. Kastner. Splitting, Renaming, Removing: A Study of Common Cleaning Activities in Jupyter Notebooks. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 114–119. IEEE, 2021.

[9] H. Fangohr, M. Beg, M. Bergemann, V. Bondar, Brockhauser, et al. Data Exploration and Analysis with Jupyter Notebooks. 2019.

[10] M. A. Jamil, M. Arif, N. S. A. Abubakar, and

A. Ahmad. Software Testing Techniques: A Literature Review. In *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, pages 177–182. IEEE, 11/22/2016 - 11/24/2016.

[11] M. Källén and T. Wrigstad. Jupyter Notebooks on GitHub: Characteristics and Code Clones. *The Art, Science, and Engineering of Programming*, 5(3), 2021.

[12] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers. The Story in the Notebook. In R. Mandryk, M. Hancock, M. Perry, and A. Cox, editors, *Engage with CHI*, pages 1–11, New York, USA, 2018. The Association for Computing Machinery.

[13] A. P. Koenzen, N. A. Ernst, and M.-A. D. Storey. Code Duplication and Reuse in Jupyter Notebooks. In M. Homer, editor, *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–9, Piscataway, NJ, 2020. IEEE.

[14] E. Matthes. *Python Crash Course: A Hands-On, Project-Based Introduction to Programming.* No Starch Press, New York, 3rd edition edition, 2023.

[15] T. Mondal, S. Barnett, A. Lal, and J. Vedurada. Cell2Doc: ML Pipeline for Generating Documentation in Computational Notebooks. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 384–396. IEEE, 2023.

[16] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire. A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories*, pages 507–517, Piscataway, NJ, 2019. IEEE.

[17] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire. Understanding and Improving the Quality and Reproducibility of Jupyter Notebooks. *Empirical software engineering*, 26(4):65, 2021.

[18] L. Quaranta. Assessing the Quality of Computational Notebooks for a Frictionless Transition From Exploration to Production. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ACM Digital Library, pages 256–260, New York, USA, 2022. Association for Computing Machinery.

[19] L. Quaranta, F. Calefato, and F. Lanubile. Eliciting Best Practices for Collaboration with Computational Notebooks. *Proceedings of the ACM on Human-Computer Interaction*, 6(CSCW1):1–41, 2022.

[20] F. Rowe, G. Maier, D. Arribas-Bel, and S. Rey. The Potential of Notebooks for Scientific Publication, Reproducibility and Dissemination. *REGION*, 7(3):E1–E5, 2020.

[21] A. Rule, A. Birmingham, C. Zuniga, I. Altintas, S.-C. Huang, R. Knight, N. Moshiri, M. H. Nguyen, S. B. Rosenthal, F. Pérez, and P. W. Rose. Ten Simple Rules for Writing and Sharing Computational Analyses in Jupyter Notebooks. *PLoS Computational Biology*, 15(7):e1007007, 2019.

[22] A. Rule, I. Drosos, A. Tabard, and J. D. Hollan. Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):1–12, 2018.

[23] F. M. Sallabi and S. Lazarova-Molnar. Teaching Modeling, Simulation, and Performance Evaluation Course Online with Jupyter Notebook: Course Development and Lessons Learned. In *2022 IEEE Frontiers in Education Conference (FIE)*, pages 1–8. IEEE, 2022.

[24] M. Schröder, F. Krüger, and S. Spors. Reproducible Research is more than Publishing Research Artefacts: A Systematic Analysis of Jupyter Notebooks from Research Articles, 2019.

[25] J. Singer. Notes on Notebooks: Is Jupyter the Bringer of Jollity? In S. Kell, editor, *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ACM Digital Library, pages 180–186, New York, USA, 2020. Association for Computing Machinery.

[26] N. Sompairac, P. V. Nazarov, U. Czerwinska, L. Cantini, A. Biton, A. Molkenov, Z. Zhumadilov, E. Barillot, F. Radvanyi, A. Gorban, U. Kairov, and A. Zinovyev. Independent Component Analysis for Unraveling the Complexity of Cancer Omics Datasets. *International journal of molecular sciences*, 20(18), 2019.

[27] L. Studtmann, S. Aydin, and H. Lichter. Histree: A Tree-Based Experiment History Tracking Tool for Jupyter Notebooks. In *Proceedings of the 30th Asia-Pacific Software Engineering Conference (APSEC 2023)*, Seoul, Korea, 2023.

[28] K. Subramanian, N. Hamdan, and J. Borchers. Casual Notebooks and Rigid Scripts: Understanding Data Science Programming. In M. Homer, editor, *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–5, Piscataway, NJ, 2020. IEEE.

[29] S. Titov, Y. Golubev, and T. Bryksin. ReSplit: Improving the Structure of Jupyter Notebooks by Re-Splitting Their Cells, 2021.

[30] G. van Rossum, B. Warsaw, and A. Coghlan. PEP 8 – Style Guide for Python Code. `https://peps.python.org/pep-0008/`, Accessed 20 November, 2023.

[31] A. P. S. Venkatesh, J. Wang, L. Li, and E. Bodden. Enhancing Comprehension and Navigation in Jupyter Notebooks with Static Analysis, 2023.

[32] J. Wagemann, F. Fierli, S. Mantovani, S. Siemen, B. Seeger, and J. Bendix. Five Guiding Principles to Make Jupyter Notebooks Fit for Earth Observation Data Education. *Remote Sensing*, 14(14):3359, 2022.

[33] J. Wang, L. Li, and A. Zeller. Better Code, Better Sharing. In G. Rothermel and D.-H. Bae, editors, *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, pages 53–56, New York, USA, 2020. ACM.

[34] J. Wang, L. Li, and A. Zeller. Restoring Execution Environments of Jupyter Notebooks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1622–1633. IEEE, 2021.