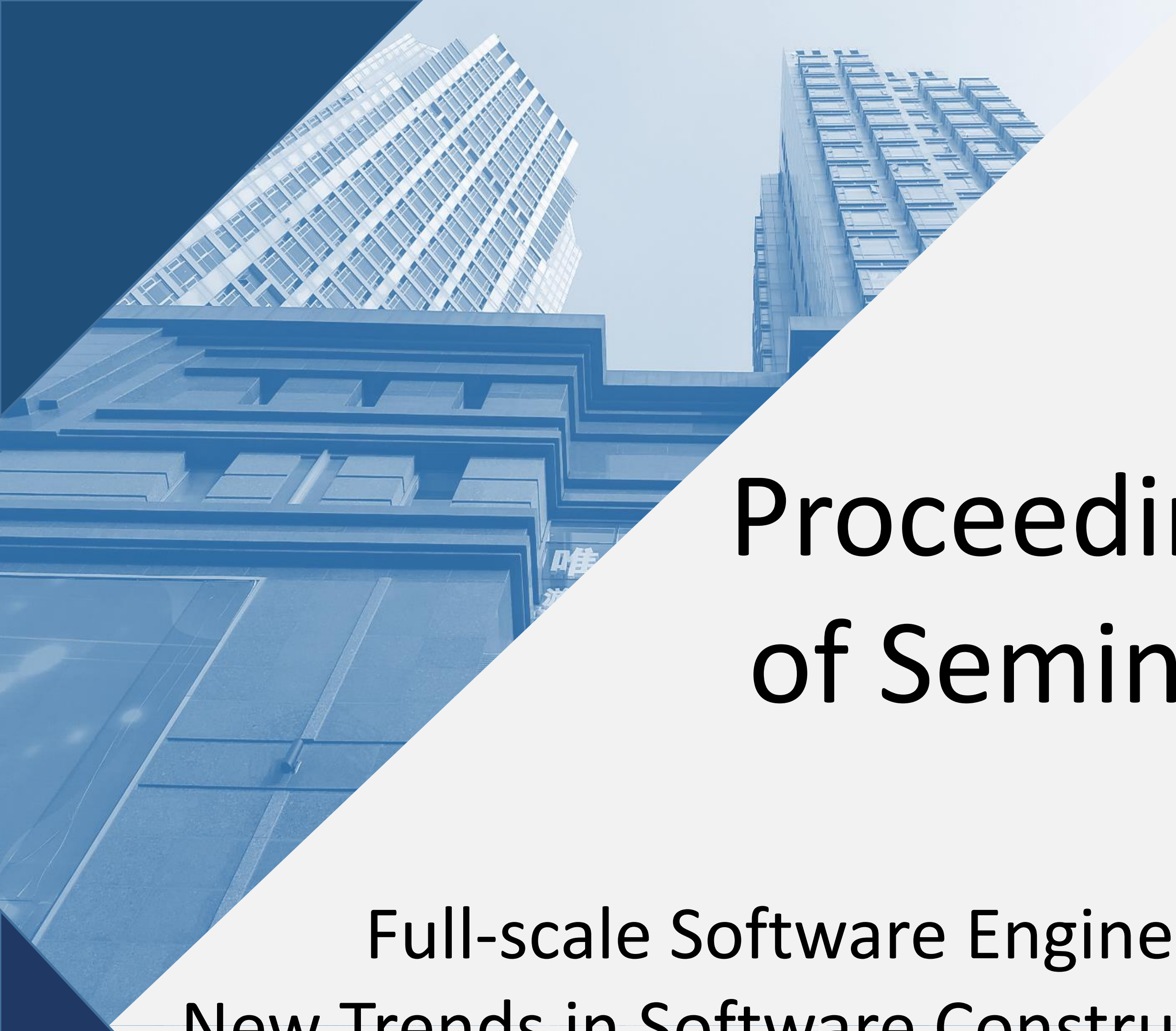


Proceedings of Seminars

Full-scale Software Engineering
New Trends in Software Construction

2021

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

SWC Software Construction

RWTH AACHEN UNIVERSITY

Table of Contents

50 Years of Mutation Testing - Has it gained any Influence on Software Testing in Practice?

Jakob Drees
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
jakob.drees1@rwth-aachen.de

Erdzan Rastoder
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
rastoder.erdzan@rwth-aachen.de

ABSTRACT

Automated software testing is an indisputable best practice during development. To assess the quality of the test suite in use, various coverage techniques are applied. For decades, the field of mutation testing has seen very active research and some advances in the applicability of the technique at an industrial scale. Mutation testing describes the practice of seeding the source code with small changes and testing whether the test suite is granular enough to catch these changes. In research, this has been viewed as the most effective way to evaluate a test suite. In this paper, we first introduce the concept of mutation testing itself and the main challenges it faces at an industrial scale. Next, we present some possible solutions suggested in the literature to tackle these challenges. Then we take a look at some industrial case studies using mutation testing and tools supporting that process that have come up over the last few years. With this, we give a brief overview of the current state of mutation testing at different scales. In the end, we give some suggestions on how its industrial adoption could be improved in the future, based on literature suggestions and the progress that has already been made.

Keywords

Mutation Testing, Software Testing, Quality of Tests

1. INTRODUCTION

A widely used testing technique to compute the quality of a test suite is code coverage. Code coverage only checks if pieces of a software's code are executed by the appropriate tests. Which does not necessarily check if the covered lines of code comply with the desired behavior. This is where mutation testing seeks to improve. As the name suggests, mutation tests automatically modify pieces of a program where tests like unit tests exist. After the program has been modified, the existing tests run on the modified program to check if the introduced modifications can be detected. Two cases can arise. The first when a mutation is detected by a test and the second when a mutation was not detected. In the latter, this might indicate that the introduced mutation is either equivalent to the original code or that the existing tests are not sufficient to discover these mutations. The mutation testing technique is considered one of the best approaches to measure the effectiveness of an existing test suite [2, 15, 17]. However, the benefits come at a price. Compared to traditional mutation testing approaches, two main disadvantages arise [4]: The first one is the sheer number of mutants that are generated when executing a mutation test. With a large amount of produced mutants comes the computational effort, which also means that not a negligible amount of time has to be used to execute the mutation tests [7]. Highlighting, the fact that the area of mutation testing was largely studied in academics focusing on small projects. Secondly, an important aspect is how mutation testing should be adopted by developers to increase the effectiveness of the existing test suite. Remarkable is, also the fact that most developers never heard or were played with the concept of a mutation testing [41]. To take up these problems, we aim in this paper to analyze solutions from large software companies like Facebook [4] or software used in safety-critical systems. Thereby, we try to find out the feasibility of using mutation testing in the industrial sector instead of the limited academic studies focusing on small projects.

1.1 Motivation

Hamelt [16] and DeMillo et al. [8] originally proposed the idea of mutation testing to test the adequacy of a test suite. Elaborating further on mutation testing, Offutt [27] supported further mutation testing ideas, that simple faults introduced by for example mutations detected by test suites, would also be able to detect faults consisting of multiple faults, which can be seen as complex faults. Despite, the potential benefits of integrating mutation testing into an existing project, its usage has mainly been limited to the academic sector [19]. In the context of industrial usage of mutation testing, it is often stated that by its usage there is struggling [36, 21]. One major problem seems to be the associated high-performance needs that have to be incorporated in a project [26]. Due to the lacking integration of tools, this issue can not be improved. From the user's point of view, again the missing integration of tools, which ease the usage of mutation testing usage, are reported [26]. While these problems are addressed in the literature, numerous effort have been done to improve the industrial usage possibility. We aim to provide information on how mutation testing is used in industry cases and highlight their used tools and methodologies. After we have analyzed existing use case scenarios, we aim to provide possible suggestions on how efficiently mutation testing could be integrated into an existing project. However, even before that, we will discuss in which industry contexts mutation testing is feasible to be applied.

2. MUTATION TESTING

The concept of mutation testing is not a new one. Therefore, much research has been directed at its strengths and weaknesses. This chapter aims to provide a short overview of the ideas behind the technique and how it compares against code coverage.

2.1 Central Ideas

Mutation testing is used to evaluate the ability of a given test suite to find faults in the code. This means it tests the test suite instead of the actual source code. This works by modifying the written source code with well-defined atomic changes, so-called mutation operators [8]. The research distinguishes between first-order mutants (FOM) and higher-order mutants (HOM), with FOM making a single change to the source code and HOM being the connection of several FOM to form a more complex fault [18]. Examples of FOM and HOM can be seen in Listing 1 and Listing 2.

Listing 1: First-Order Mutant

```
while (x > y):
    x = x / 10;
return x

Can be mutated to:

while (False):
    x = x / 10;
return x
```

Listing 2: Higher-Order Mutant

```
while (x > y):
    x = x / 10;
return x

Can be mutated to:

while (y >= x):
    y = x / 10;
return y
```

Because of the vast number of possibilities, there are a lot of slightly different mutation operators.
The method of mutation testing has been suggested in the late 1970s by DeMillo et al. [8] and bases on two basic hypotheses:

1. Competent programmer hypothesis: This hypothesis assumes that programmers write sound and efficient code that is nearly perfect and contains only small errors. Therefore, a program could also be corrected with only relatively few changes since the code works as intended for the most part.
2. Coupling effect hypothesis: The coupling effect describes that test data that can detect very simple errors in a program also must be able to detect more complex programming errors because it is already sensitive enough to find these simple errors. With simple errors meaning something like an error in an assignment. This is derived from the thought that more complex errors are just compounds of simple errors.

After a mutation operator has been applied to the source code of the program, the test suite is run on the mutated code. Now, there are two possible outcomes: The first one being one or more tests fail, killing the mutant and therefore showing the test suites resistance to that mutation. The second outcome would be that all tests still pass, which means the mutant survives. Surviving mutants are hints for code that is not well tested. Once a mutant survives, it has to be evaluated if the mutant points to not well-tested code or if it is equivalent [30]. We speak of an equivalent mutant when it is a mutated version of the original program that is equivalent to the original source and therefore indistinguishable for the test suite. These equivalent mutants should not be considered further since they do nothing to improve the tests [8]. Recently, a slightly different approach has been suggested by Petrović et al.[35] that looks at productive versus unproductive mutants instead of sheer equivalence. With this distinction, equivalent mutants can be considered productive as well, because they can improve the source code itself rather than the test code. This can happen when the equivalent mutant achieves the same result, but maybe with an improved encapsulation or more concise code. Unproductive, however, means mutants that are not useful because they are equivalent to the original source and do not drive source code improvement or point to places in the code that should often not be tested, such as the exact string of an error message. Since this is not a task trivially automated, developers have to do much of this evaluation effort.
After reviewing the unkilled mutants, a mutation score can be calculated from the number of killed mutants divided by the number of non-equivalent mutants. This mutation score then quantifies the mutation resistance or adequacy of the current test suite. The reviewing developers can of course also decide to write additional tests to kill some surviving mutants. After the addition of those tests, the mutation test has to be applied again to verify the results and calculate the new mutation score. To calculate a meaningful mutation score, a lot of different mutation operators need to be applied to various lines of the source code.
Again, it is important to understand that mutation testing is used as a test evaluation rather than a source code evaluation. In that sense, it is best compared to coverage metrics that are widely applied in software development. Nevertheless, a programmer might decide to rewrite parts of the source code because of an equivalent mutant showing a more elegant and equivalent way.

2.2 Mutation Testing vs. Code Coverage

In software testing, the usage of various coverage criteria for assessing the test suite is fairly widespread [43]. It has a very similar goal to mutation testing, in terms of evaluating the test instead of the source code itself. Code coverage divides the program into syntactical structures and then tracks which test executes which part of the code. For example, statement coverage is a metric for the percentage of source code lines visited by the test suite of that program. This way, a test suite with high coverage is perceived as better (to a certain extent) compared to one with a lower coverage score, since its theoretical ability to find faults in the program is much higher. This explanation makes clear that statement coverage in particular and code coverage, in general, is a syntactical technique of evaluating the test suite. Its quantitative approach indicates that a higher coverage hints for more intense testing. However, a big problem of all coverage notions is that they are easily tricked because they are of a syntactical nature. Imagine, for example, a test that calls a method but never asserts a value or asserts something always true (i.e. 1 == 1) as seen in Listing 3. This test would cover at least some lines of that method, but not detect any faults.

Listing 3: Weak test

```
function test_some_logic():
    x = some_logic()
    assert 1 == 1
```

Similarly, just by chance, the chosen test data sometimes might not cover certain edge cases that would lead to an error but still, the called code would count as fully covered. As shown in research, this leads to the coverage metrics being an imperfect metric for assessing a test suite [17] [24].
Since mutation testing follows a completely different approach, it is not tricked that easily. It works directly on the source code and mutates statements in a (semi-)random way, meaning that its evaluation is done semantically rather than syntactically. Also, there is strong evidence that mutations are a valid substitute for real faults in a program, making them extremely effective at seeding faults into not well-tested parts of the source code [20, 34]. Therefore, this technique is more likely to find not fully tested parts than coverage criteria. This has been shown by various studies in the past [22, 32].

2.3 Main Challenges

Mutation testing is in its pure form a very thorough technique, mutating almost every line and seeding multiple possible mutation operators per line. Many studies found mutation testing to be a very effective technique to evaluate and improve the quality of the test suite [34, 41, 20]. Also, a lot of research has been directed at finding mutation operators that provide value and resemble real faults that can be found by a test suite [9, 18]. Its thoroughness leads to mutation testings superior evaluation of a test suite but at the same time places two major burdens on the development process:

1. Extremely high computational effort needed
2. Time-intensive review process for developers

Firstly, because of the sheer number of possibilities, a program can be mutated, the first step of mutation testing takes a long time. The examples in Listing 1 and Listing 2 should indicate that there are infinitely many ways to "mutate" a program. For every mutation, the test suite needs to be executed in order to assess if any tests fail. This quickly adds up to a strong load on the testing infrastructure. As calculated by Klischies and Fögen [21] the application of mutation testing can take up around 60 times the time the normal test suite would take, making it seem infeasible for larger projects. Additionally, even though research has been directed at reducing the number of unproductive mutants [9, 40, 3] and therefore prevent false alarms for the developers, still, the developers have to go through each surviving mutant manually and decide if it is a defect in the test suite or rather an unimportant equivalent mutation. This can take up much time in the development process, potentially slowing it down tremendously.
Lastly, it should be noted that despite the strong suggestion of researchers that mutation testing should be seen as a gold standard for test suite assessment, developers seem not to be convinced that it is worth the time they potentially would have to invest in the review process. This, of course, is a major hindrance since developers are the ones who have to use this workflow. Nevertheless, this problem is of course highly correlated with the other two, since developers should be more open to applying mutation testing when the overall overhead that comes with it is relatively small.

3. SUGGESTED APPROACHES TO MUTATION TESTING WEAKNESSES

This chapter aims to present some suggested approaches to overcome these challenges in practice and therefore be able to apply mutation testing without the tremendous overhead it is often associated with.

3.1 Reducing the Computational Effort

1. **Random and Selective Selection**
   The goal to reduce the computational effort of executing mutation tests, which is closely related to the number of mutants generated, is to reduce the produced mutants. One simple way is to choose on a random basis which mutants should be included. Even though, leaving some mutants out, Malevris and Papadakis [31] state that selecting an exemplary 20% of the mutants randomly causes a 7% fault loss, which they argue is a reasonable trade-off. Additionally, based on large open-source projects, Gopinath et al. [13] found out that randomly selecting a constant number of selected mutants statistically results similar to the case when using all possible mutants.
   To highlight the differences between selecting mutants randomly or from the selective mutation, Zhang et al. [44] came to the conclusion, that no important differences can be seen between these two approaches. Whereas Gopinath [14], even though, with the same conclusion of the non-significant difference between the two approaches, also elaborated a maximal improvement over the random selection of mutants, which is reached at 13%. These numerous works, seem to provide enough evidence, on having the safe option to choose randomly mutants to reduce their amount, while not significantly impacting its capabilities.

2. **Selection based on Importance**
   Another approach to reducing the number of generated mutants is to scope the mutants to the available budget and time which a project has. This is done by

ranking the mutants according to their perceived importance. This allows the actual testers to configure their analysis so that only the most important mutants are used. Based on a set of already analyzed mutants, Sridharan et al. [42] by using a Bayesian methodology select by prioritization the most informative mutants. These informative selected mutants give the testers the possibility to concentrate on the mutants which require the most work to kill them.

3.2 Time Intensive Review Process

Once the mutants have been generated and run against the current test suite, a set of mutants that survived the test suite is left. One of the most important goals is to identify the surviving mutants that only resemble code that is equivalent to the initial source code, and therefore probably not of great use in enhancing the test suite. As mentioned in section 2, a new notion of productiveness has been introduced, where even equivalent mutants can be productive because they might trigger the developer to rewrite parts of the code to be more elegant or efficient. Nevertheless, the bulk of research in the past has been focused on reducing equivalent mutants and not differentiating between productive and unproductive mutants, which only came up relatively recently.
For the most part, it is obvious that reducing the computational effort and reducing the time needed for a review process will have similar goals, namely dramatically reducing the number of mutants that are generated and therefore, tested. Another common goal would be to detect the equivalence of mutants with the original source code early on, so they would not have to be reviewed and maybe not even tested against each other.
The former has already been discussed in the last section. However, the latter has seen a lot of progress with the use of HOM. A great advantage of these HOM is that it is relatively easy to evaluate if a mutant is equivalent. One problem with their use is the creation of these special mutants [18]. It is a difficult problem to find relevant combinations of first-order mutants that represent an effective HOM and places an additional burden on the preparation of the mutation testing process. But it is still an active research field and has shown promising results recently [28].
Another approach to detecting equivalent mutants early on is a technique called ”Trivial Compiler Equivalence” (TCE). With TCE for a large number of mutants, it can be decided efficiently if they are equivalent or not [29]. This greatly reduces the number of mutants that have to be tested and since it eliminates equivalent mutants, it specifically reduces the time that developers have to invest into the review process because it strips out a large portion of unnecessary mutants [7].
Another important aspect of improving the review process is how and when developers are presented with the surviving mutants. However, the research on this has been fairly shallow to non-existent. Most case studies try to integrate it in their normal code review process while the tests run, they are displayed to the developer and any reviewers that might have to give the code a pass. An approach taken by Petrovic and Ivankovic [33] was to surface only a very limited subset of surviving mutants at a time to keep the impact on the review process as little as possible. This greatly reduces the number of mutants that can be acted upon, but tries to achieve a balance between usability during development and improving the mutation score.

3.3 Low Adoption in Software Development

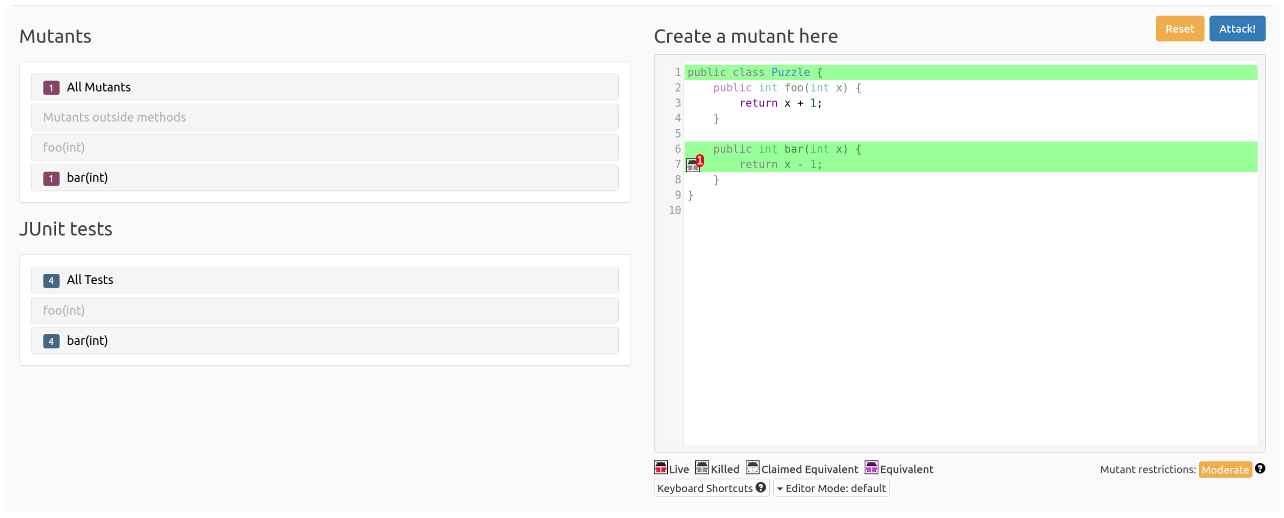
Arguably, one of the most important aspects to drive mutation testing adoption is to convince software developers of its use. Even though there does not seem to be reliable data on this topic, some case studies show that most developers in the industry are not aware of its existence in the first place, let alone that it is seen as the gold standard for test evaluation in research [4] [7]. But first and foremost, this gold standard comes with a high cost that not every developer sees as worth for the use they are getting. Additionally, it is extremely difficult to prove the effect of mutation testing in a specific program because to do so one would have to maintain two versions. One that works with mutation testing, the other one with the usual workflow, and then check whether the version that has been developed and released with the usual workflow contains the same faults as the mutation tested one. There has been some research on these specific problems but still, this is seen as a hindrance [34].
As pointed out in section 2, the doubts that developers have regarding the effect they get for the extra effort is highly influenced by the overhead mutation testing that will actually add to their development process, while the computational effort should not play a big role. Therefore, advances in reducing the time for the mutant review process and the number of surfaced mutants already has a positive impact on mutation testing evaluation.

Apart from case studies, an approach to drive adoption of mutation testing has been the development of the game CODE DEFENDERS[1] by Rojas and Fraser [38]. In this game, two players, an attacker and a defender try to seed mutations into source code or respectively write a strong test suite to find the faults (see Figure 1). This gamification of mutation testing seems to be very interesting, especially for new developers just learning the concepts of testing.

4. MUTATION TESTING IN PRACTICE

The introduction of mutation testing often struggles in the software industry [25]. Madeyski et al. state that due to several issues, the struggle is enforced [23]. In an experiment to integrate mutation testing in existing source code Nica et al. [26] confirmed the statements of a large number of the generated mutants and the problem of having multiple equivalent mutants, which cause the high computational effort problem. Furthermore, the necessary time to integrate mutation testing to set up the experiment seems extremely high [25], considering that the setup consists of configuring the tool which conducts the mutation testing technique to execute the mutation testing, took several months. Based on the necessary time to execute mutation tests on Math4, which includes 850 tests, it was estimated that to execute mutation testing on Apache Solr, which on its part has 10500 tests, it would take approximately 540 hours to run the test single threaded [25]. With the assumption that the test coverage lies at 90%, whereby the time to execute the mutation test would increase with lower test coverage. This again confirms the first stated problems of the great computational effort, which are needed to run the mutation tests. Despite the effort which are necessary to integrate mutation testing inside

[1]`https://code-defenders.org`



```java
public class Puzzle {
    public int foo(int x) {
        return x + 1;
    }

    public int bar(int x) {
        return x - 1;
    }
}
```

Figure 1: In CODE DEFENDERS, a player can try to create mutants by hand, trying to avoid a certain test suite. Or write that test suite themselves to kill mutants.

a company's existing software suite, we present successful integrations.

4.1 Mutation Testing at Facebook

One of the most recent examples of large-scale mutation testing is a case study that has been conducted at Facebook [4] within their mobile code base, only focusing on the parts that are written in Java. The goal of the case study was to evaluate whether mutation testing could be introduced at Facebook without causing a tremendous overhead for both the developers and the infrastructure. To achieve this, they developed a tool called "Mutation Monkey" that is responsible for introducing mutations into source code submitted by developers to the version control system. These mutants are then subject to the same testing process the normal commit would be and at the end, a surviving mutant can be presented to the developer via their usual tool for code review. It is important to point out that the normal process for a code commit at Facebook already contains several steps to reduce the computational effort for testing (e.g. greatly reduce the number of executed tests with the help of a previously trained program). During the initial development of Mutation Monkey, its developers used internal and external data sources for Java programming faults and the respective fix to greatly reduce the number of mutation operators applied by Mutation Monkey while still maintaining the usefulness of mutation testing. Included in these data sources were for example past crashes of the mobile app and the corresponding code faults that led to these crashes. The rationale behind this learning of fewer mutants was that they would be better suited for the specific codebase at hand, rather than seeding numerous mutations that would not have great potential to show important flaws in the testing strategy. When put into production, Mutation Monkey used 19 different mutation operators. Additionally, Mutation Monkey tracks which tests have executed but not killed the specific mutant to improve the quality of the results and give the developer a better understanding of the tests that could potentially be improved.
Over the period of two months, 15,000 mutants had been produced with a survival rate of 60 - 70% depending on the mutation operator. The authors of the study point out that this is an extremely high survival rate when compared to the baseline of around 10 - 15% when using conventional mutation testing in comparable research [33]. Especially, since these mutations are by design not equivalent to the original program. Still, it is important to note that the survival rate of a mutant can only serve as an indicator, but not as proof of its usefulness. During the period of the research, the application of Mutation Monkey also did not produce noticeable infrastructure costs because of the reduced number of mutation operators and the scheduling of their computation which took place during less busy hours infrastructure-wise. To evaluate how developers saw the introduction of Mutation Monkey, the researchers also conducted several interviews with developers that had worked on the mobile Java code and therefore been subject to the Mutation Monkey testing. Somewhat interesting is that none of the 26 developers interviewed knew about the concept of mutation testing beforehand. This again points to the extremely low adoption in the industry. After being introduced to it and working with some of Mutation Monkeys' suggestions, most of the developers thought of it as a good idea and the majority of this group took the surfaced mutant as a reason to improve a test or write a new one. But, even though the mutants produced by Mutation Monkey were based upon past production crashes and other common Java source code faults, some developers still rejected the idea that mutation testing would give benefits in the day-to-day development process, arguing that one could alter a program in infinitely many ways and writing a test for all of them was just not feasible. The paper did not give any information on how long it took developers to evaluate whether a mutant can be used to improve the test suite. Nevertheless, the authors stress that a large portion of the interviewed developers decided to act upon the surfaced mutants and therefore showing the usefulness of learned mutation operators in practice.

4.2 Mutation Testing in Safety Critical Software

To evaluate the effectiveness of using mutation testing in the industry, Delgado-Perez et al. conducted an experiment where a software suite written in C provided by a nuclear industry was selected to apply the mutation testing technique [7]. In [6] also a safety-critical program was used to find out if the introduced mutations correlate with real code faults, where 85% of the mutations have also been produced by real faults. Additionally, in [1] it was concluded that the mutations which were injected by a mutation test were more comparable to real code faults when comparing with mutations introduced willingly instead of using a mutation testing tool. To avoid the high computational resources necessary to execute a mutation test, the experiment only used a subset of mutation operators which reduces the number of produced mutants, while assuming that the chosen subset represents also the missing one. Even when reducing the number of produced mutants there might still be a duplicate, equivalent, or invalid mutants present which are also called ineffective. Therefore, TCE is used to remove these ineffective mutants. As for a possible mutation testing tool that would be used, an open-source and free tool was chosen to highlight the possibility of using mutation testing without paid commercial solutions. As a tool Milu 3.2 [2] was used which natively supports TCE and as for the subset of operators, 12 out of the 77 present operators were selected. After executing the mutation tests on the selected safety-critical nuclear software the number of generated mutations for each tested function was considerably lower, compared to other studies, ranging from 75 to 331 generated mutants. These represent the mutants before the application of TCE, where after its execution nearly 50% were detected as ineffective and therefore ignored. Surprisingly, the remaining mutants almost all equal valid faults at 96%. As for the effort to augment the existing test suite and for the replacement of the branch test suite, only around 20% have been added additionally to the existing test suite. This leads to the researchers' conclusion that the necessary effort to provide to include mutation testing in safety-critical software suites is plausible to achieve.

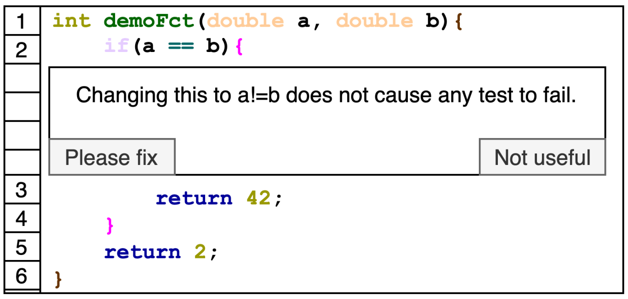
4.3 Tools used in Industry

Tool for executing Mutation Testing

PITest is a mutation testing framework for one of the most used programming languages, Java [10]. An important practical feature of PITest is the good integration with common tools used in the industry for development like build tools or development environments [5]. Additionally, to cope with the possible performance impact of having to generate a lot of mutants, PITest does not work with the actual source code but uses the byte code. Additionally, to analyze the results of the mutation process, PITest generates a report which allows to easily switch between the results and the concerned code sections.

When common development tools are used, the integration is straightforward as it merely requires the addition of a task in the build file to configure PITest like which classes should be tested or defining the output directory of the results. Afterward, the build file can be used as usually as PITest will not make any modifications to the compiled code nor will it leave any artifacts, except the generated reports which are saved in the defined directory.

[2]Milu `https://github.com/yuejia/Milu`



```
int demoFct(double a, double b){
    if(a == b){
```

Changing this to a!=b does not cause any test to fail.

```
        return 42;
    }
    return 2;
}
```

Figure 2: Mutation Review Tool

Tools for Monitoring Mutation Testing

When programming collaboratively in the industry on a common project the review process is an important one as also stated by Linus' Law [37] "Given enough, eyeballs, all bugs are shallow". Indirectly indicating that the practice of reviewing committed code from other developers is one option to ensure the absence of bugs in the reviewed code. Now if the integration of mutation testing should be adopted inside a company that already has these workflows of the review process, it would be beneficial to integrate the mutation also inside the review process. Similar to the possibility of leaving comments on regular sections of code, which should be resolved by the original author of the written code, Google has similarly integrated the review of mutation testing in their code review process [33]. Exemplary in figure 2 we see a demonstration scenario that can occur during the review process. We notice that the reviewer is presented with a dialog presenting more information about the mutation which has been done on that precise line of code. In this case, the mutation has not been noticed by a test. Now the reviewer has the choice to decide if this mutation should be noticed by an additional test or if it would not be useful to write an additional test for that specific mutation. In the first case, when the reviewer clicks on the "Please fix" button, the original author of the code gets notified that an additional test should be written, to catch this mutation. In the second case, the reviewer clicks on the "Not useful" button, whereupon no one gets notified to improve the code for that specific mutation. Petrovic et al. state in [33] that the code review process is the ideal place to integrate mutation testing for newly written code, as the submitted code is reviewed and thus, can be marked as further improvable, increasing the possibility to find bugs and minimize them in the final product. Sadowski et al. state in [39] that the integration of the mutation test analyzation results inside the developer's workflow is a key point to improve their effectiveness. On contrary, if the results have to be looked at in a different workflow, thus ripping the developers out from their usual to look at the mutation test results, their effectiveness is drastically reduced. The huge number of possible mutants per line of code that can be generated, together with the fact that not all lines of code should be mutated as lines used for logging, Google in [33] uses a probabilistic mutation testing analysis technique and one limiting the generated mutants to only interesting lines of code, to limit these disadvantages. To limit the generated mutants, Google takes the approach of only generating one mutant per line of code. Otherwise, with numerous generated mutants the review process would suffer, as the review interface gets cluttered and the actual reviewers will get quickly confused. Therefore, the mutation operator is picked randomly from the possible mutants which can be picked for that possible mutation environment. However, this randomly picking of a mutation operator loses its randomness over time, as with the information of the reviewers who decide if a mutation is considered useful or not. Google integrates these decisions into the probabilistic mutation generation process over time to improve the selection of a "random" mutant. As for the detection of relevant lines of code where mutations should be generated. Google utilizes an arid node detection via abstract syntax tree traversal, described in [33]. With these improvements to limit the number of generated mutants and to limit their generation to lines of code that should be mutated, the review process's effectiveness gets improved by avoiding unnecessarily included mutations.

5. HOW TO FURTHER DRIVE MUTATION TESTING ADOPTION

The results presented in section 4 show very promising approaches to the application of mutation testing at industry scale. The insights by some larger software companies and a growing number of tools to make the application process easier are keys to that. In this section, we want to build on these approaches and propose some ideas on how mutation testing could be used in the future.

5.1 When and how to use mutation testing

As shown in subsection 3.1 and section 4 there exist numerous ways to decrease the computational effort that can be successfully applied to real-world projects while still preserving most benefits of mutation testing. With this being said, nowadays, the time developers have to invest in evaluating mutants is a far larger burden than the computational effort placed on the infrastructure. When it is still not possible to apply mutation testing to the whole codebase because of the computational overhead, it should be considered to only test certain key parts to ensure the adequacy of the test suite when it comes to the core of the application. This is especially true when the business logic is held within only a few components.
Furthermore, neglecting the size of the software project, mutation testing should be used when software failures could potentially harm human beings or cause tremendous financial loss. A safety-critical system, as presented in subsection 4.2 would be the perfect example for such a case. The failure of its computer systems could lead to incredible and lasting damage to its surroundings, making a rigid test suite essential.
Additionally, while it might be interesting to apply mutation testing to the entire codebase of an existing project, this will probably not produce the most effective results, since the amount of surfaced mutants is most likely far too high to be handled properly. Instead, it is more effective to apply the technique with every change that is made to the software and therefore improving the test quality little by little.
A very promising looking approach to both decrease computational effort and reduce the number of equivalent mutants has been to use learned HOM that are derived from past faults [4]. Surely, this requires additional effort when introducing mutation testing but could pay off in the long term, making it a viable option for larger long-term projects. Additionally, it might be possible to create a number of these learned HOM for different languages and make them publicly available for future use in various projects with that specific language. This would require several repositories with common bugs and their respective fixes to learn these in the first place but would help smaller projects to use these more advanced techniques.
To take this even further, one could imagine that the creation of such a repository could also be structured by use case or system type. The rationale behind this is that different types of systems often employ totally different architectures and therefore probably benefit from mutation testing in very different ways. By capturing the nature of those systems, it might be possible to tailor mutations for various system types driving its effectiveness and efficiency.

5.2 Tackling developer doubts

A large problem remains with the low awareness of developers and their doubts regarding the application of mutation testing. We argue that both should be addressed in tandem by introducing developers to the technique and immediately showing the benefits. The most obvious place for such an introduction would be universities. Here, students learn software development and engineering, including how to test their code. A great addition to this repertoire would be the application of mutation testing in small example projects. This way, students would become familiar with the concepts and also experience their usefulness firsthand in a potentially less goal-oriented environment than a software company trying to hold a deadline. Additionally, we suggest that mutation testing should be added to the curriculum of professional certifications like the ones from ISTQB[3]. The International Software Testing Qualifications Board (ISTQB) provides different certifications for software testers, none of which contains the practice of mutation testing at the moment. Adding mutation testing could lead to raising awareness among software testers and therefore drive adoption independently of the universities.
While the scenarios presented above would be the ideal long-term scenario, it is not very likely to come before a broader adoption of mutation testing in the industry, and therefore developers already working have to be convinced first. This poses a different challenge because of the many different companies and projects software developers work at. Large tech companies such as Facebook and Google can play an important role in proofing that mutation testing can be applied at a large scale and acting as kind of a blueprint for smaller companies. In that sense, past and future case studies of these two and how mutation testing improved their test suite are a great way to drive adoption. Nevertheless, it is unlikely that this alone will suffice. Instead, the cost of applying mutation testing should be driven as low as possible to allow adoption for smaller teams without the need to invest hours into the process. An example could be the automatic generation of test code after a mutant surfaced, as shown by Fraser and Zeller [12]. This takes a lot of the work away from the developer but poses a more difficult challenge to researchers. Additionally, this might not be possible for a large portion of surfaced mutants because of

[3]`https://www.istqb.org/`

the often complex nature of source code. In these cases, Facebook's approach is promising because it provides the developer with context information about which test cases visited the mutated part of the code but failed to detect it. This way, the amount of time for developers to evaluate the usefulness of the mutant is greatly reduced. It shows how crucial the review process is for adoption by software developers.

The game CODE DEFENDERS, as presented in subsection 3.3, is also very promising. Gamification has shown great success in other areas and might help a great deal in making developers aware of the technology. It should be noted, however, that the game might not be best suited for promoting the unique strengths of mutation testing since it only shows the creation and killing of mutants rather than how the process could be made easier for a developer or how it improves the test suite specifically. We would like to see further developments in this field, as they are already happening [11]. One could for example add additional languages to make it more accessible to a broader audience than only Java developers.

Lastly, it might be beneficial to conduct an experiment similar to Facebook's case study [4] to drive mutation testing acceptance of software developers. The authors of that paper argued that even though they could not prove the real use of mutation testing regarding the prevention of actual production faults, they saw it as a success that developers deemed the information useful and improved the test suite as a result. While this might be a difficult thing to pull off since developers often choose and design the tools they use, it can still be a way of showing the use of mutation testing to developers who would have considered the effects non-existent.

5.3 Improving the Mutant Review Process

As already highlighted in subsection 4.3 an effective strategy to integrate mutation testing inside the existing workflow of the developers is to use the code review phase. Similar to Google, Facebook integrated the generation of mutants to be shown as code changes to the developer. Having these two large companies emphasize the usefulness to integrate mutation testing also in the code review shows the possible importance for companies that would like to integrate mutation testing in their software. However, it should be noted that both Facebook and Google integrated mutation testing in their code review workflow and systems. Even though Facebook integrated the workflow in their code review tool Phabricator, new companies trying to use mutation testing and similarly would like to integrate mutation testing in their code review workflow would need additional setup work. For larger companies that do have the capacities, this would be feasible, but the question remains if smaller ones would have the same possibilities to have a dedicated developer team at the beginning of the mutation testing integration. In the case of integrating the generated mutations in the code review process, it would be important as stated in [33, 4] to maximize the generation of useful mutations as much as possible. As otherwise, the code review, which still largely involves developers to review the code changes, would suffer from ineffective or equivalent generated mutants. Thereby, also affecting their effectiveness and causing confusion[33]. To mitigate the disadvantages of generating numerous mutations having no effect on the code, tools limiting the generated mutants are applicable. The first step into this direction would be like in [33] to randomly select an applicable mutant operator and thus limiting the possible generated mutants to one for each code line. This would greatly help small companies which try to quickly integrate mutation testing as they do not have to deal with the huge number of generated mutants. The positive side of using this approach is that it can be enhanced. If a company sees how useful mutation testing is, additional features can be added to improve the random picking of mutation operators. Like in the case of Google [33] the information gathered from the review phase can be analyzed to switch from the random picking to a probabilistic one. This would be an upgrade that companies with enough capacities and motivation to use mutation testing could integrate. The same applies to highlighting which lines of code should be considered in the mutants' generation phase. Like lines where logging functions are written, which would only clutter up the code review phase [4].

6. CONCLUSION

Mutation Testing is still an evolving discipline that can enhance software development to find its way more and more in the industry. Despite the disadvantages surround mutation testing, numerous progressions have been made in this direction. We presented possible tools which allow teams in a project to integrate mutation testing in their development workflow. This integration is the first step to improve its usage inside a team. Regarding the computational aspects of mutation testing, simpler and more elaborated methods have been proposed to reduce computational costs. These methods range from randomly selecting mutants to complex mutants' recognitions, implemented in existing tools. Leading to the presentation of existing tools which actually perform the mutation testing, which can be used in existing projects. Based on our results, we believe that the integration of mutation testing is a feasible option if a software project must assure with more rigorous their software project. Nevertheless, this path is not limited to industry partners with high capacities, as also small developer teams can benefit from a moderate workload using the proposed approach from mutation testing.

7. REFERENCES

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In G. Roman, W. G. Griswold, and B. Nuseibeh, editors, *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 402–411. ACM, 2005.

[2] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.

[3] D. Basile, M. H. ter Beek, M. Cordy, and A. Legay. Tackling the equivalent mutant problem in real-time systems: the 12 commandments of model-based mutation testing. In R. E. Lopez-Herrejon, editor, *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*, pages 30:1–30:11. ACM, 2020.

[4] M. Beller, C. Wong, J. Bader, A. Scott, M. Machalica, S. Chandra, and E. Meijer. What it would take to use mutation testing in industry-a study at facebook. *CoRR*, abs/2010.13464, 2020.

[5] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. PIT: a practical mutation testing tool for java (demo). In A. Zeller and A. Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 449–452. ACM, 2016.

[6] M. Daran and P. Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. In S. J. Zeil and W. Tracz, editors, *Proceedings of the 1996 International Symposium on Software Testing and Analysis, ISSTA 1996, San Diego, CA, USA, January 8-10, 1996*, pages 158–171. ACM, 1996.

[7] P. Delgado-Pérez, I. Habli, S. Gregory, R. Alexander, J. A. Clark, and I. Medina-Bulo. Evaluation of mutation testing in a nuclear industry case study. *IEEE Trans. Reliab.*, 67(4):1406–1419, 2018.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[9] V. Do, Q. Nguyen, and T. Nguyen. Evaluating mutation operator and test case effectiveness by means of mutation testing. In N. T. Nguyen, S. Chittayasothorn, D. Niyato, and B. Trawinski, editors, *Intelligent Information and Database Systems - 13th Asian Conference, ACIIDS 2021, Phuket, Thailand, April 7-10, 2021, Proceedings*, volume 12672 of *Lecture Notes in Computer Science*, pages 837–850. Springer, 2021.

[10] O. Ezenwoye. What language? - the choice of an introductory programming language. In *IEEE Frontiers in Education Conference, FIE 2018, San Jose, CA, USA, October 3-6, 2018*, pages 1–8. IEEE, 2018.

[11] G. Fraser, A. Gambi, and J. M. Rojas. Teaching software testing with the code defenders testing game: Experiences and improvements. In *13th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2020, Porto, Portugal, October 24-28, 2020*, pages 461–464. IEEE, 2020.

[12] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Trans. Software Eng.*, 38(2):278–292, 2012.

[13] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce. How hard does mutation analysis have to be, anyway? In *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*, pages 216–227. IEEE Computer Society, 2015.

[14] R. Gopinath, M. A. Alipour, I. Ahmed, C. Jensen, and A. Groce. On the limits of mutation reduction strategies. In L. K. Dillon, W. Visser, and L. A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 511–522. ACM, 2016.

[15] R. Gopinath, C. Jensen, and A. Groce. Code coverage for suite evaluation by developers. In P. Jalote, L. C. Briand, and A. van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 72–82. ACM, 2014.

[16] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Software Eng.*, 3(4):279–290, 1977.

[17] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In P. Jalote, L. C. Briand, and A. van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 435–445. ACM, 2014.

[18] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), 28-29 September 2008, Beijing, China*, pages 249–258. IEEE Computer Society, 2008.

[19] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[20] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In S. Cheung, A. Orso, and M. D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 654–665. ACM, 2014.

[21] D. Klischies and K. Fögen. An analysis of current mutation testing techniques applied to real world examples. *Full-scale Software Engineering/Current Trends in Release Engineering*, 13:52, 2016.

[22] N. Li, U. Praphamontripong, and J. Offutt. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009, Workshops Proceedings*, pages 220–229. IEEE Computer Society, 2009.

[23] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Trans. Software Eng.*, 40(1):23–42, 2014.

[24] B. Marick. How to misuse code coverage. 04 2000.

[25] J. Mozucha and B. Rossi. Is mutation testing ready to be adopted industry-wide? In P. Abrahamsson, A. Jedlitschka, A. Nguyen-Duc, M. Felderer, S. Amasaki, and T. Mikkonen, editors, *Product-Focused Software Process Improvement - 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings*, volume 10027 of *Lecture Notes in Computer Science*, pages 217–232, 2016.

[26] S. Nica, R. Ramler, and F. Wotawa. Is mutation testing scalable for real-world software projects? In *VALID Third International Conference on Advances*

in System Testing and Validation Lifecycle, Barcelona, Spain, 2011.

[27] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, 1992.

[28] S. Oh, S. Lee, and S. Yoo. Effectively sampling higher order mutants using causal effect. *CoRR*, abs/2104.11005, 2021.

[29] M. Papadakis, Y. Jia, M. Harman, and Y. L. Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In A. Bertolino, G. Canfora, and S. G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 936–946. IEEE Computer Society, 2015.

[30] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman. Chapter six - mutation testing advances: An analysis and survey. *Adv. Comput.*, 112:275–378, 2019.

[31] M. Papadakis and N. Malevris. An empirical evaluation of the first and second order mutation testing strategies. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, Workshops Proceedings*, pages 90–99. IEEE Computer Society, 2010.

[32] A. Parsai and S. Demeyer. Comparing mutation coverage against branch coverage in an industrial setting. *CoRR*, abs/2104.11767, 2021.

[33] G. Petrovic and M. Ivankovic. State of mutation testing at google. In F. Paulisch and J. Bosch, editors, *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 163–171. ACM, 2018.

[34] G. Petrovic, M. Ivankovic, G. Fraser, and R. Just. Does mutation testing improve testing practices? *CoRR*, abs/2103.07189, 2021.

[35] G. Petrovic, M. Ivankovic, B. Kurtz, P. Ammann, and R. Just. An industrial application of mutation testing: Lessons, challenges, and research directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*, pages 47–53. IEEE Computer Society, 2018.

[36] S. Rani, B. Suri, and S. K. Khatri. Experimental comparison of automated mutation testing tools for java. pages 1–6, 09 2015.

[37] E. S. Raymond. *The cathedral and the bazaar - musings on Linux and Open Source by an accidental revolutionary.* O'Reilly, 1999.

[38] J. M. Rojas and G. Fraser. Code defenders: A mutation testing game. In *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016*, pages 162–167. IEEE Computer Society, 2016.

[39] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In A. Bertolino, G. Canfora, and S. G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 598–608. IEEE Computer Society, 2015.

[40] D. Schuler and A. Zeller. Covering and uncovering equivalent mutants. *Softw. Test. Verification Reliab.*, 23(5):353–374, 2013.

[41] B. H. Smith and L. A. Williams. On guiding the augmentation of an automated test suite via mutation analysis. *Empir. Softw. Eng.*, 14(3):341–369, 2009.

[42] M. Sridharan and A. S. Namin. Prioritizing mutation operators based on importance sampling. In *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*, pages 378–387. IEEE Computer Society, 2010.

[43] D. Tengeri, Á. Beszédes, D. Havas, and T. Gyimóthy. Toolset and program repository for code coverage-based test suite analysis and manipulation. In *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*, pages 47–52. IEEE Computer Society, 2014.

[44] L. Zhang, S. Hou, J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 435–444. ACM, 2010.

An Overview of Fault Taxonomies For Test Automation

Md Tasin Siddiqi
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
tasin.siddiqi@rwth-aachen.de

Timo Rohrer
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
timo.rohrer@rwth-aachen.de

ABSTRACT

Fault taxonomies have been developed to aid in understanding root causes as well as to get an understanding of the diversity of faults in software projects. The fault taxonomies range from general to application specific, with the goal to support quality assurance activities. Testing the faults can occur manually or with automation, however making the distinction which to choose is difficult. This paper gives a background to fault taxonomies and testing in general, and discusses one general and one JavaScript specific taxonomy in more detail. For the different categories of faults identified, the applicable testing techniques as found in relevant research are discussed, such as automated regression or manual black box testing. Thus, an overview of which categories of faults in taxonomies are best automated or tested manually are identified to develop a cohesive testing strategy.

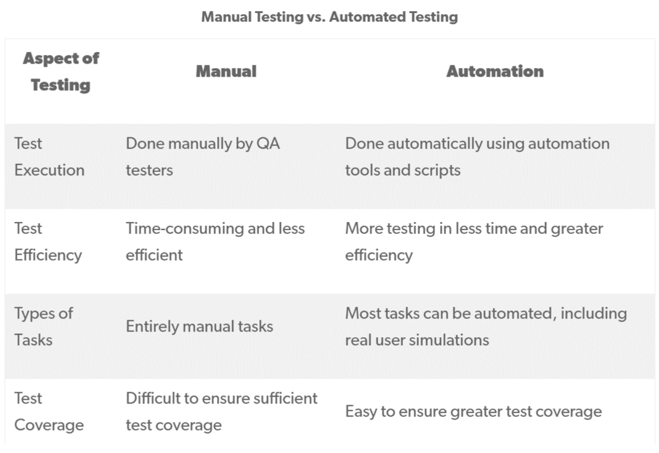
Keywords

Fault Taxonomy, Test Automation, Testing Strategy

1. INTRODUCTION

In this paper titled 'Fault Taxonomies to Drive Test Automation', we discuss the well documented taxonomies of faults that prevail in industry and why they do in the first place. Software testing plays a integral role in the Software Development Lifecycle, even from the earliest days of computers [17, 18]. Every software goes to several iterations of testing before it is first released for customers, since it is "crucial to the success of a software project" [5]. While not every project or application might necessitate exhaustive testing, there are many examples of software where testing is crucial and a minor bug might result in loss of millions of dollars (as in the case of financial software) or worse loss of human lives (in case of healthcare [37] or military [34] software). Even if all software might not be designed for such precision or to be responsive to the millisecond, all contemporary software should be tested to ensure data security and minor bugs because of which the user loses interest and ends up not using the software [3]. As there are many alternatives for a particular software genre in this competitive world, more and more companies are investing and innovating to ensure that their software are optimized to capture the largest share of customer base. This can only be possible if the software is flawless or at least close to it. Therefore test engineers and product managers are constantly coming up with testing protocols that are comprehensive and capture a large taxonomy of bugs which can be fixed at the right time and with as few resources as possible. Striking this balance requires careful consideration of which faults should be accounted for by manual testing and which should be covered by an automated test suite. And while the creation of fault taxonomies in academia and industry projects is plentiful [30, 9, 22, 25, 27, 39], there has been little consideration in how to leverage the taxonomies to help decide what to automate.

There is no one correct universal method of testing and most often devising a suitable test strategy is an art than a science which requires great expertise specific to a niche industry. There is a constant trade-off between manual testing and automation testing and frequently it comes down to the experience of a Senior Test Architect who then decides when and what to automate and what to leave out to be tested manually. However, it must be noted that there are certain standard frameworks and test suites for certain genre of applications for which a more objective decision can be made.


Manual Testing vs. Automated Testing

Aspect of Testing	Manual	Automation
Test Execution	Done manually by QA testers	Done automatically using automation tools and scripts
Test Efficiency	Time-consuming and less efficient	More testing in less time and greater efficiency
Types of Tasks	Entirely manual tasks	Most tasks can be automated, including real user simulations
Test Coverage	Difficult to ensure sufficient test coverage	Easy to ensure greater test coverage

Figure 1: Differences Between Manual and Automation Testing [24]

Research Questions

Many bug taxonomies have been developed to support quality assurance activities. The goal of this research is to get an overview of the bug taxonomies available in order to extract the essence that can be used to drive test automation. Our paper basically revolves around the following questions :

- **RQ1 :** What type of bugs can be easily detected by manual testing?
- **RQ2 :** Which are easier to detect by automated testing?
- **RQ3 :** Do we need structured test suites to detect the bug types or should we utilize explorative testing?
- **RQ4 :** What can we learn from those taxonomies in order to create a test automation strategy?

Organization

We start off in Section 2 by establishing the context and defining the lingo that is prevalent in the Test-Automation Community. Then we proceed onto describe what Testing is in general. From there we dive deep into the broad categories of Software Testing, namely : Manual Testing Methods (such as Exploratory Testing, Negative Testing and Black Box Testing) and Automated Testing (such as such as Automated benchmarking, Combinatorial testing and Automated regression testing). Next, some related work is discussed in Section 3. In section 4 two bug taxonomies are presented: a more general one by Catolino et. al. in the paper titled *Not All Bugs Are the Same* and then a more specific one by Gyimesi et. al. in the paper titled *BugJS*. In Section 5 we discuss which bug types as defined by the taxonomies can or should be automated and what should be left out to be tested manually. Lastly, in Section 6 we conclude the paper.

2. BACKGROUND

2.1 Fault taxonomies

Prevalent in the natural sciences, a taxonomy is created by defining distinct groups of things based on some shared characteristics and then aggregating them to form a classification. While the most well known application for taxonomies is biology, they have also been used widely in business, economics, and computer science. At it's core, a taxonomy categorizes or classifies things into types. Frequently, an important part of taxonomies is a hierarchical ordering, forming a tree structure. In the context of software engineering, there are multiple applications for taxonomies, including classifying tools and techniques such as a taxonomy of model-based testing approaches [36]. In the context of this paper bug taxonomies will be examined, where software bugs are classified. To understand bugs, first it is necessary to give same definitions:

DEFINITION 1 (FAILURE). *A failure is "an event that occurs when the delivered service deviates from correct service", which "means that at least one (or more) external state of the system deviates from the correct service state. The deviation is called an error" [4].*

For example, a failure might be a crash of an application, a network connection loss, or slow loading of content on a website.

DEFINITION 2 (FAULT). *"The adjudged or hypothesized cause of an error is called a fault", or bug [4].*

For example, the the previous example failure of an application crash, the underlying bug might be a segmentation fault, also called a memory access violation.

Over the years, many bug taxonomies have been proposed, ranging from system and application specific to more general ones. Some commonly defined properties for taxonomies as aggregated by Lough [26] include:

- **Comprehensible**, so that experts and those new to the field can understand and extract value out of the taxonomy
- **Exhaustive and complete**, meaning any thing must be able to "fit" in the structure and all possible categories are accounted for
- **Deterministic and objective**, so that the characteristics by which a thing is classified by can be automatically "extracted" from the thing only and be "clearly observable"
- **Mutually exclusive**, meaning "categories must not overlap"
- **Primitive and repeatable**, meaning the classification would be the same if repeated by another party
- **Specific and unambiguous**, meaning established and well defined terminology is used

2.2 Testing

Software Testing is the process of testing the quality of software in order to deliver software applications of the highest quality to the client. There are software applications that require great precision and a minute fault may result in a catastrophe: the loss of millions of dollars or the loss of human lives. Therefore a structured method which tests software at each stage of development becomes crucial to ensure that the release software is devoid of any such bugs. Over the years many strategies and approaches of software testing have been developed to achieve this goal. We shall discuss the most common and widely used approaches that exists in industry these days [29, 6].

2.2.1 Manual testing

In this type of testing, the software testers manually execute test cases without the use of any automation tools. They simulate the role of the end-user and try to find as many bugs as possible. The bugs are finally reported into a report which are passed on to developers to fix them. Manual testing often focuses on usability, performance testing, and assessing the overall software quality. Among the most common manual testing methods are :

- Exploratory Testing : is a type of dynamic testing, which allows the tester freedom to interact with the application as they like without a predefined test procedure.

- Black Box Testing : is more structured, where an application's functionality is tested based on the requirements or specifications it was built for. The inner workings of the application are not considered in this type of testing.
- Negative Testing : is when invalid inputs are inserted to check if the application handles exceptions properly and returns errors appropriately. This type of testing validates that the application can handle improper usage.
- Usability Testing : is a method that does not rely in writing test cases in code, rather it evaluates an application by having users interact with it. Additional reviewers watch the users and gather feedback on the ease of use of the application.
- Acceptance Testing : is a method where, frequently the intended end user of the application, tests it to ensure the solution will work for their needs.
- Manual Regression testing : is a method of verification which is used to confirm that a recent update, bug fix or code change has not adversely affected the pre-existing features. It re-executes some of the test cases to ensure that existing functionality works correctly and no new bugs have been introduced.
- Test Case Execution : Test cases help the tester through a sequence of steps to validate whether the application is working as intended. A good test case requires good writing skills, attention to detail and a good understanding of the application.

2.2.2 Automated Testing

In this type of testing an automation tool is used to execute pre-scripted test cases. The objective of automation testing is to increase productivity of testers by simplifying and increasing efficiency in the testing process [32, 33]. Among the advantages of Automated Testing over Manual testing are the following:

- Quicker : Automated testing speeds up the process of software testing in comparison to manual testing. In terms of testing execution, it will increase productivity and reduce testing time for the majority of apps/websites.
- Avoids Repetition : Repetitive tasks are inefficient when done manually, especially when they reoccur. There is also an increased chance of human error. Automated testing can eradicate this, depending on the quality and scope of the test cases.

While automated testing has a lot of advantages, it has is disadvantages as well such as :

- High setup cost : Initial set up costs (automation tool purchase, training and tutorials, maintenance of test scripts) are expensive. Also, if the app or website changes regularly, the cost and time associated with script maintenance will increase considerably.
- Not Universal : While automated testing is great for types of testing like stress testing and smoke testing, it's not suited for everything. Looking at user interface, documentation, installation, compatibility, and recovery are often better suited to manual tests.

3. RELATED WORK

Among the earliest work on developing fault taxonomies for software was by Chillarege et al. [11] in the form of the Orthogonal Defect Classification. In this taxonomy, categories such as Algorithm or Assignment defects were identified. More research by Freimut et al. [16] developed a taxonomy which focused more on the orgin of the fault. More root cause based taxonomies were developed by Chan et al. [10]. Some of the categories include parameter incompatibility fault, misunderstood behaviour fault, response faults, and time-out exceptions. Work on more application specifc taxonomies include one on JavaScript conducted by Hanam et al. [20]. Fault types included missing arguments, unhandeled exepctions, dereferenced non-values, and incorrect comparisons. For Java applications, Hovemeyer at al. [21] defined a number of bug patterns, including Redundant Comparison to Null, Dropped Exception, Null Pointer Dereference, and Inconsistent Synchronization.

4. FAULT TAXONOMIES

In the following section two bug taxonomies will be presented, a general one by Catolino et al. [8] as well as a more specific one by Gyimesi et al. [19]. For each, an overview of the taxonomy is given first and then descriptions of the bug types identified which are relevant to this research are discussed.

4.1 Not All Bugs Are the Same

This paper by Catolino et al. [8] aims to identify and classify reported bugs thereby building a taxonomy of bugs so that the process of bug triage, that is, the process of assigning the fixing of a reported bug to the most qualified developer. The basis of classification is identifying the root cause of the reported bugs. This paper found nine main common root causes of bugs during the study.

Research Methodology

The empirical study definition and design that were followed in this paper to create a bug root cause taxonomy. The research questions (RQ) formulated in the study were:

- **RQ1** To what extent can bug root causes be categorized through the information contained in bug reports?
- **RQ2** What are the characteristics, in terms of frequency, topics, and bug fixing time, of different bug categories?
- **RQ3** How effective is our classification model in classifying bugs according to their root cause exploiting bug report information?

Configuration issue

Example summary.

"JEE5 Web model does not update on changes in web.xml"
[Eclipse-WTP Java EE Tools] - Bug report: `190198`

Figure 2: Example of Configuration issue bug report [8]

The first taxonomy of bug root cause happen to be configuration issue. It is a taxonomy mainly related to a wrong usage of external dependencies that cause issues in the web model of the application. It mostly occurs due to external dependencies such as external libraries when some software package is installed for the first time.

Network issue

Example summary.

"During a recent reorganization of code a couple of weeks ago, SSL recording no longer works"
[Eclipse-z_Archived] - Bug report: `62674`

Figure 3: Example of Network issue bug report [8]

This is the category of bugs having connection or server issues as it's root cause. It is caused by network problems, server shutdowns or communication protocols that are not properly used within the source code. The server side network issues are probably more in the control of the testers/troubleshooters than the client side one. A network issue could range from a total loss of connectivity to intermittent connectivity and even performance problems.

Database related issue

Example summary.

"Database connection stops action servlet from loading"
[Apache Struts] - Bug report: `STR-26`

Figure 4: Example of Database issue bug report [8]

This category of bugs comprise all connection issues between the database and the main application. An example of this type of issue are issues related to failed queries. All bugs related to wrong SQL statements are part of this category because they relate to issues in the communication between the application and an external database, rather than characterizing issues within the application.

GUI related issue

Example summary.

"Text when typing in input box is not viewable."
[Mozilla-Tech Evangelism Graveyard] - Bug report: `152059`

Figure 5: Example of GUI issue bug report [8]

This category includes all bugs related to the Graphical User Interface of a software project. It comprises all issues referring to stylistic errors, i.e. screen layouts, element colors and padding, text box appearance, and buttons as well as all un-usual messages appearing to the user.

Performance issue

Example summary.

"Loading a large script in the Rhino debugger results in an endless loop (100% CPU utilization)"
[Mozilla-Core] - Bug report: `206561`

Figure 6: Example of Performance issue bug report [8]

This category comprises all bugs that are related to performance issues, including memory overuse including memory leaks and methods with infinite loop.

Permission/Deprecation issue

Example summary.

"setTrackModification(boolean) not deprecated; but does not work"
[Eclipse-EMF] - Bug report: `80110`

Figure 7: Example of Permission issue bug report [8]

This category includes all bugs related to presence, modification or removal of deprecated method calls or APIs. Also problems related to unused APIs are included in this category.

Security issue

Example summary.

"Disable cocoon reload parameter for security reasons"
[Apache-Lenya] - Bug report: `37631`

Figure 8: Example of Security issue bug report [8]

All issues related to vulnerability and security are part of this category. The kind of bugs related to this category are those such as reloading certain parameters and removal of unused permissions that might decrease the overall reliability of the system.

Program Anomaly Issue

Example summary.

"Program terminates prematurely before all execution events are loaded in the model"
[Eclipse-z_Archived] - Bug report: `92067`

Figure 9: Example of Anomaly issue bug report [8]

Bugs introduced by developers when trying to enhance existing source code and that are concerned with specific circumstances such as exceptions, problems with return values and unexpected crashes arising due to error in the logic of the program (rather than GUI related problems) are all part of this category of issue. Bug reports in this category tends to generally have the source code included as well so that a discussion around possible fix can be accelerated.

Test Code-related issue

Example summary.

"[the test] makes mochitest-plain time out when the HTML5 parser is enabled"
[Mozilla-Core] - Bug report: 92067

Figure 10: Example of Test code issue bug report [8]

This category comprises of bugs appearing in test code. Bugs in this category report problems related to test cases.

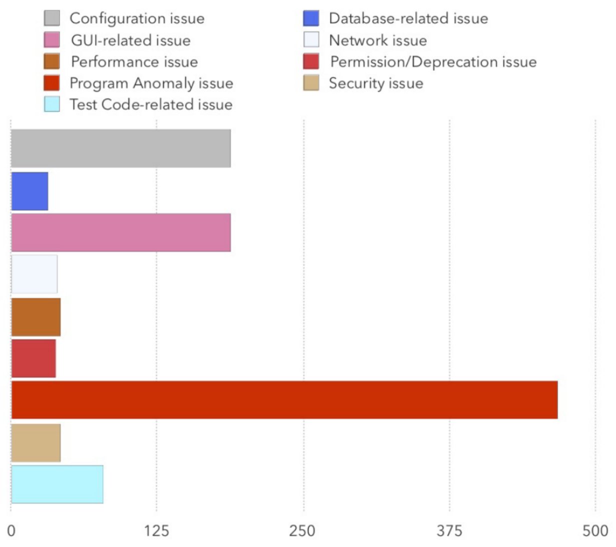


Figure 11: Frequency of each category of bug root cause [8]

4.2 BugsJS

An example of a more application specific fault taxonomy is BugsJS by Gyimesi et al. [19]. Ten server side JavaScript libraries powered by Node.js, including Express and Mongoose, were selected after a survey of open source projects on Github. On their Issues pages, bugs were selected based on criteria such as the bug-fixes being low complexity and in isolation as well as relevant test cases being included. In total, 542 bugs and their corresponding fixes and test cases were selected for the final dataset. Note: example code may be shortened for better comprehension.

As seen in Figure 12, the authors of the paper identified four major categories of bugs: incomplete feature implementation, incorrect feature implementation, generic errors, and perfective maintenance. Perfective maintenance refers to feature enhancements and therefore will no be considered in this paper.

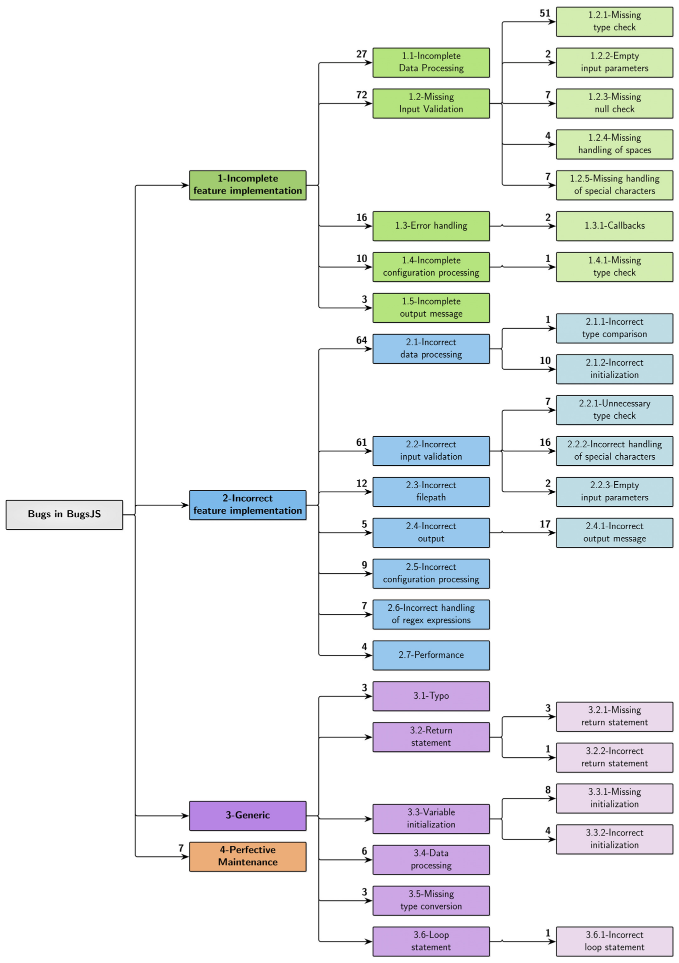

Bugs in BugsJS
1-Incomplete feature implementation
27
1.1-Incomplete Data Processing
72
1.2-Missing Input Validation
51
1.2.1-Missing type check
2
1.2.2-Empty input parameters
7
1.2.3-Missing null check
4
1.2.4-Missing handling of spaces
7
1.2.5-Missing handling of special characters
16
1.3-Error handling
2
1.3.1-Callbacks
10
1.4-Incomplete configuration processing
1
1.4.1-Missing type check
3
1.5-Incomplete output message
2-Incorrect feature implementation
64
2.1-Incorrect data processing
1
2.1.1-Incorrect type comparison
10
2.1.2-Incorrect initialization
61
2.2-Incorrect input validation
7
2.2.1-Unnecessary type check
16
2.2.2-Incorrect handling of special characters
2
2.2.3-Empty input parameters
12
2.3-Incorrect filepath
5
2.4-Incorrect output
17
2.4.1-Incorrect output message
9
2.5-Incorrect configuration processing
7
2.6-Incorrect handling of regex expressions
4
2.7-Performance
3-Generic
3
3.1-Typo
3.2-Return statement
3
3.2.1-Missing return statement
1
3.2.2-Incorrect return statement
3.3-Variable initialization
8
3.3.1-Missing initialization
4
3.3.2-Incorrect initialization
6
3.4-Data processing
3
3.5-Missing type conversion
3.6-Loop statement
1
3.6.1-Incorrect loop statement
7
4-Perfective Maintenance

Figure 12: Taxonomy of bugs in the benchmark of JavaScript programs of BugsJS [19]

- Incomplete feature implementation refers to cases where the specification for a feature was not fully considered in the implementation. According to the authors, these cases frequently occurred when corner-cases appeared or when requirements changed over time, exposing the underlying incomplete implementation.
- Incorrect feature implementation bugs are a wrong implementation of the desired feature, frequently due to an incorrect interpretation of the requirements.
- Generic errors differ from the previous category in that they arise from common programming mistakes, not a misunderstanding of the requirements.

Incomplete feature implementation

Incomplete Data processing

A large category of bugs identified by the authors is incomplete data processing. These bugs occur when an incorrect interpretation of how the input should be transformed causes an error. An example can be seen next, where a helper function from the Lodash library needed to be added to properly escape special characters.

```
- var text = $(this).html();
+ var text = _.escape($(this).text());
```

Missing input validation

An example of such an error is shown below, where an additional condition was added to only execute the operation if a line number exists.

```
- if (file && file.sourceMap) {
+ if (file && file.sourceMap && line) {
```

Error handling

The category of bugs identified as error handling are a perfect fit for manual negative testing. Incomplete error handling occurs when the software does not fully cover all exceptions. An example can be found below, where before the program did not properly throw an error.

```
- procWrapper.on('SIGINT', disconnectBrowsers)
+ procWrapper.on('SIGINT', function () {
+     disconnectBrowsers(process.exitCode)
+ })
```

Incomplete configuration processing

Errors which pertain to incomplete configuration of the parameters to a a program fall in this category and below can be seen a example from Karma.

```
- options[helper.toCamel(name)] = argv[name];
+ options[helper.toCamel(name)] = argumentValue;
```

Incomplete output message

Any output messages that were incomplete fall in this category. In the example below from Hessian, any number which exceed the maximum safe integer are casted to a string so that they could be safely read.

```
+ if (val.greaterThan(MAX_SAFE_INT)) {
+   val = val.toString();
```

Incorrect feature implementation

Incorrect data processing

A large percentage of bugs found in the dataset belong to the category Incorrect data processing, referring to a fundamental error in the logic of data transformation. An example from Eslint can be seen below, where it needs to be decided if a rename is necessary or not. If the property lacks a key, it is not necessary, so a additional condition was added to this if statement.

```
- if (props[i].computed) {
+ if (props[i].computed || !props[i].key) {
```

Incorrect input validation

Another large category of bugs includes errors in checking if a given input is valid. An example from Eslint can be seen below, where a function was causing false negatives on octal numeric literals.

```
- /^[0-9]+$/.test(source.get(node.obj).value))
+ astUtils.isDecimalInteger(node.obj))
```

Incorrect filepath

A category of bugs which fixes incorrect filepaths was identified by the authors. A bug from Hexo can be seen below, where the path can be both absolute or relative and is then resolved.

```
- if (path[0] !== '/') path = '/' + path;
- return config.url + path;
+ return urlFn.resolve(url || config.url, path);
```

Incorrect output

Bugs where the output was not correct were grouped in a category. A Bug from Eslint can be seen below, where instead of the last, the penultimate item should be returned.

```
- loc: lastItem.loc.end,
+ loc: penultimateToken.loc.end,
```

Incorrect configuration processing

Any bugs which relate to the incorrect configuration of the values of parameters to a program fall in this category. An example from Eslint can be seen below, were unnecessary default options were passed even when not explicitly specified.

```
- let parserOptions = Object.assign(
-   {}, defaultConfig.parserOtpions);
+ let parserOptions = {};
```

Incorrect handling of regex expressions

Bugs related to incorrect regex expression were grouped in one category. An example from Eslint can be seen below, where a a check for multiple spaces had to be fixed.

```
- const multi = /( {2,})+?/,
+ const multi = /( {2,})( [+*{?]|[^+*{?]|$)/,
```

Performance

As a part of incorrect feature implementation, the authors identify performance as a category of bugs which occur. These bugs are frequently discovered from normal usage of the application, since an error caused a large amount of computer resources to be used, such as memory or CPU time. An example is a bug is Eslint where a regex caused catastrophic backtracking, because it was matching non-linebreak characters followed by a linebreak. The new regex shown below simply matches newlines.

```
+ const lineEnd = /\r\n|[\r\n\u2028\u2029]/g;
```

Generic

Typo

A common error type that developers might make are typos, as identified in the taxonomy. Below is an example from Pencilblue, where the word prototype was spelled incorrectly.

```
- BODY_PARSER_MAP[mime] = protoype;
+ BODY_PARSER_MAP[mime] = prototype;
```

Return statement

Forgetting a return statement was another category of bugs. Below is an example from Karma.

```
- msg = config.formatError(msg)
+ return config.formatError(msg)
```

Variable initialization

Either incorrectly initializing a variable or missing it entirely are in this category of bugs. An example from Mongoose can be seen below.

```
- var values;
+ var values, errorMessage;
```

Data processing

Generic data processing errors were categorized in this group, and below an example can be seen from Eslint.

```
- node.id[0] === node.id[0].toUpperCase()
+ node.id[0] !== node.id[0].toLowerCase()
```

Missing type conversion

Generic missing type conversion errors were captured in this category. Below an example from Shields can be seen, where the labels of the string had to be converted to a string before changing them to upper case.

```
- data.text[0] = data.text[0].toUpper();
+ data.text[0] = ('' + data.text[0]).toUpper();
```

Loop statement

The last section in the category of generic bug is an incorrect loop statement, as seen in the example from Eslint below.

```
- while ((currentAnc = currentAnc.parent)) {
+ do {
     if (isConditionalTest(currentAnc)) {
         return currentAnc.parent;
     }
- }
+ } while ((currentAnc = currentAnc.parent));
```

5. DISCUSSION

In this section the two bug taxnomies are discussed to extract the gist of the different bug categories. Furthermore, for the different bug types, the best testing strategy is discussed, whether that be manual or automated testing. The recommendations are based on existing literature and also specify the sub type of testing, for example exploratory testing or regression testing.

Automated database testing

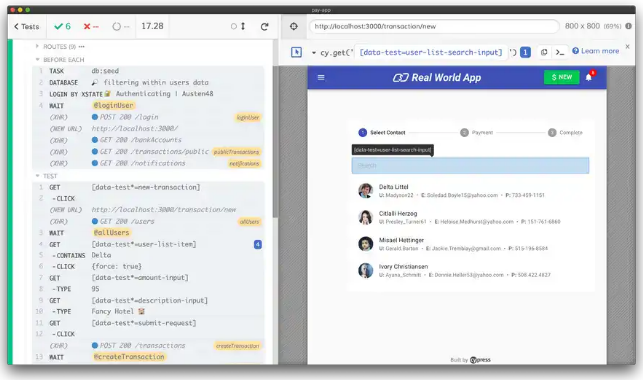
Database related issues were only identified in the more general bug taxonomy, since the BugsJS taxonomy does not cover a system that includes databases. However, they is certainly a category of bugs that can be automated. With increasing data complexities, heterogeneous environments and data sizes in terabytes, it is really difficult to form a testing strategy based on manual testing. In these cases, test automation can help perform data validation, schema verification, database health check, security checks, etc. It helps in performing the right data checks in optimum time-frame and budget. However, to ensure there are no database crashes, failovers, broken insertions or deletions, one needs to create a sound database test automation strategy. This can be achieved by following the right steps.

Here's are the 8 steps to performing automation database testing accurately [1]:

1. *Identification of Scope* The first thing is to identify what needs to be covered in database testing, what can be excluded, and which all data sets are impacting the application performance most.
2. *Test Script Preparation* The next step is to prepare the test scripts which requires one to grab needful SQL queries to be executed, identify conditional flows and prepare automation scripts.
3. *Test Case Identification* This step is to identify the priority test cases to carry out data checks with query to query or query to previously stored data.
4. *Execution* In this step, one needs to execute scripts that are prepared or tweak them to perform as expected.
5. *Reporting* Once the execution is over we can dig into the detailed analytics and generate reports for managers and developers to take necessary actions.
6. *Post Execution Monitoring* Post the execution and reporting, monitor the test results to identify the trends and make necessary tweaks and fixes to the test scripts.
7. *Test Script Reuse* Post the unit execution, you can further reuse the test script for regression test execution.
8. *Cross-checking with UI Testing Report* Lastly, cross-check the database automation testing results with UI testing and affirm the findings made by two separate channels to take further actions.

Frameworks such as Selanium can be extended to provid automatic databases testing functionality. The authors in [12] report a significant reduction of 92% in the effort to execute automated tests in databases when compared to manual execution using such a tool.

Automated GUI testing



```
ROUTES (9)
BEFORE EACH
TASK            db:seed
DATABASE        filtering within users data
LOGIN BY XSTATE Authenticating | Austen48
WAIT            @loginUser
(XHR)           POST 200 /login                      loginUser
(NEW URL)       http://localhost:3000/
(XHR)           GET 200 /bankAccounts
(XHR)           GET 200 /transactions/public         publicTransactions
(XHR)           GET 200 /notifications               notifications
TEST
GET             [data-test*=new-transaction]
 -CLICK
(NEW URL)       http://localhost:3000/transaction/new
(XHR)           GET 200 /users                       allUsers
WAIT            @allUsers
GET             [data-test*=user-list-item]          4
 -CONTAINS      Delta
 -CLICK         {force: true}
GET             [data-test*=amount-input]
 -TYPE          95
GET             [data-test*=description-input]
 -TYPE          Fancy Hotel
GET             [data-test*=submit-request]
 -CLICK
(XHR)           POST 200 /transactions               createTransaction
WAIT            @createTransaction
```

Figure 13: Example of the test status menu from automated GUI testing application Cypress [2]

GUI related issues were also not mentioned in the BugsJS taxonomy since it only pertains to server side libraries, but they are another type of bugs that can be automated. In fact there already exists tools such as Selenium, Cypress, AutoIT, iMacros, Watir, EggPlant etc. to perform GUI testing. After a command line interface (CLI) has been substituted with the graphical user interface, developers start to face drawbacks. It turns out that the experience of using a graphical front-end is different depending on a device's screen dimension, operating system, browser of choice, and so on. Graphical user interface testing allows development teams to ensure that users can experience all the elements of the website in the same way regardless of their platform or browser. As studies demonstrated, "GUI testing is an applicable technology for automated system testing with effort gains over manual system test practices" [7]. For example, as seen in Figure 14, authors in [13] showed that the initial effort to setup GUI testing automation is higher, but it proves to more efficient over time.

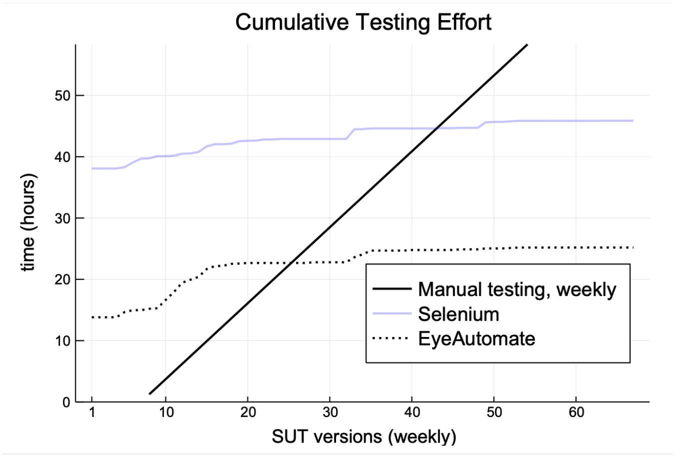


Figure 14: The cumulative testing effort over time for weekly manual testing and Selenium and EyeAutomate [13]

However, there may still be errors in the visual representation of an application than automated test suites would have difficulty catching. So while automation might cover most cases, some edge defects may still need to be manually checked.

Automated performance testing

A category of bugs which exists in both taxonmies and that can be tested in an automated manner is *Performance*. While the argument might be made that egregious performance issues will almost always be caught by simply using the application, it is still valuable to build an automated performance testing infrastructure to benchmark the application. Relying on subjective experience is not a consistent enough approach to make sure the performance of software stays satisfactory. In addition, the prevalence of mature frameworks to benchmark code makes performance testing easily automated. Well written tests should be agnostic of the implementation, meaning little rewriting of tests is required. After refactoring a large part of the application, automated benchmarking can verify the changes did not introduce performance issues. Libraries for JavaScript such as Benchmark.js as well as Webload for more general testing can be used to easily write statistically significant benchmarking tests. Automated performance testing checks the speed, response time, reliability, resource usage, and scalability of software under an expected workload by leveraging automation [38]. Within performance testing, there are a number of different sub categories. Load testing tests the performance by increasing the load until it reaches a threshold. Stress testing tests the performance and stability when hardware resources aren't sufficient. This type of testing is also applicable to the similar *network issues* as defined in the general taxonomy.

Automated regression testing

For verifying that changes to an application do not regress the state of them, automated regression testing has proven a valuable tool. As discussed by authors in [31], "since tests ensure compliance to the behavior coded in them, deviations from this behavior will cause affected tests to fail. This capability of tests is cited to increase developers' confidence to refactoring code". Depending on the scope and coverage of the tests, automated regression testing can catch *program anomaly issues* as defined by the more generic taxonomy. Here, the category concerns errors that arise from "enhancing existing source code", such as refactoring activities, and automated regression testing is a perfect match.

Automated code analysis

Automated code analysis tools such as ESLint are a common tool to find possible code errors and deviations in the style guidelines [35] of software projects. Often a part of the automatic CI/CD workflow, code analysis tools can enforce code consistency, catch common programming mistakes, and preemptively prevent compile or runtime errors. For example, for JavaScript ESLint can enforce rules to catch errors for such bugs like *variable initialization* as defined in BugsJS. Additionally, *typos* and *missing returns* are also able to be detected by ESLint, and other code analysis tools. While, for example, not all typos can be caught by such methods, they represent an invaluable tool in a testing suite to ensure high quality code and are an obvious application for automation.

Manual black box testing

The premise of black box testing is to test compliance of the software with the requirements [28]. The code is therefore not considered in this method. This means exact specifications and requirements are required for the software tester to understand what the software should do. This leads it well suited to a number of bug types as defined in the BugsJS taxonomy. *Incomplete or incorrect outputs* are are categories of faults which are typically well defined in the feature requirements. Here, the defined requirements allow the tester to interact with the feature and write test cases without looking at the source code. As such, it is well suited to black box testing. Additionally, multiple bug categories from the more generic fault taxonomy suit this type of testing well. The category *Security issue* is best done manually, since there are many vulnerabilities that a human needs to verify [14]. Security demands are frequently well documented as part of the software development cycle to pass legal requirements such as data protection rules. With such well defined requirements, it is a good fit for manual black box testing. By an analogous argument, the category of bugs described as *Permission/Deprecation issue* is best done manually as well.

Manual negative testing

Negative testing is using incorrect or invalid inputs to "to stimulate error-handling of the system" as well as "check for appropriate responses" [15]. Therefore it is well suited to the category of bugs defined as *Error handling* in the BugsJS taxonomy. The idea is to induce error states in the application through invalid inputs and to observe the response. Exception handling using interrupts or signals or seamless error detection and correction are possible to fix missing error handling. However, to automate this type of testing would be difficult, since the whole premise is to break the software. With induced program crashes or undefined behavior, such negative testing is best done manually.

Manual exploratory testing

As opposed to black box testing, exploratory testing does not follow set guidelines. The goal is for testers to explore the application and write test cases at the same time [23]. As such, for categories of bugs which pertain more to feature implementation and that do not have well defined requirements, this type of testing is well suited. From the BugsJS taxonomy categories of bugs such as *incomplete and incorrect data processing* as well as *missing and incorrect input validation* are best dealt with by exploratory testing. Similarly, *incomplete or incorrect configuration processing* and *incorrect filepath* are errors which occur in the implementation of a feature. Generic errors like *incorrect regex expressions, type conversions* and *incorrect loop statements* are very difficult to automate. Additionally, these bug categories necessitate understanding the inner workings of the application, so black box testing is not adequate. The category of bugs which the Not All Bugs Are the Same taxonomy defines as *Test Code-related issue* is best handled by manual exploratory testing. Writing a test suite to find the bugs in the test suite is not economical, and therefore best done in a manual fashion.

Lessons from taxonomies for automation

Though there are many tools available to automate almost all genre of issues from performance to GUI issues, it is often the experience of the Test Engineers that choose to automate certain categories of bugs while testing others manually. However, as demonstrated in this paper, more objective techniques are possible to be developed. From analyzing two taxonomies, some commonalities have emerged. Performance, a category of bugs in both, is a clear example were extensive academic literature and industry practice has established it as a clear application for automation. On the other hand, faults pertaining to error handling are best handle through manual negative testing. While taxonomies are often most useful when they are specific to the project or niche they pertain too, the lessons for which type of faults can be best automated from this paper can be used as recommendations after the taxonomy is already in place. In essence, it is best to create a taxonomy that is among others; complete, deterministic, and repeatable as described in Section 2. Then, this research can be used as a starting point to figure out which of the categories of faults are to be automated. With a fault taxonomy and testing recommendations for the different fault categories, a comprehensive strategy for testing has thus been developed.

6. CONCLUSION

Software Testing is an art that requires years of experience and domain expertise to decide on a test plan strategy to be adopted. There are situations where manual testing might be superior than automated testing and again instances where automated testing is better than manual testing. However, there is never a clear cut answer. More than often, a good test strategy has incorporates both manual and automated tests. Both Manual and Automated testing should be looked at as two tools in the toolbox of a mechanic. It's the clever discretion of the mechanic when and where to use those tools to get the job done. We discuss both of those tools here in this paper in detail. We've also talked about the fault taxonomies encountered most often and referenced two papers which talk about a general bug taxonomy and then another which is more specific to Javascript. Finally we have discussed which specific testing methods suits a fault type best.

7. REFERENCES

[1] 8 Steps to Performing Automated Database Testing Accurately. https://www.testing-whiz.com/blog/8-steps-to-performing-automated-database-testing-accurately. Accessed on 2021-05-30.

[2] JavaScript End to End Testing Framework.

[3] S. S. R. Ahamed. Studying the feasibility and importance of software testing: An analysis. *CoRR*, abs/1001.4193, 2010.

[4] A. Avizienis, J. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.

[5] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Trans. Software Eng.*, 13(12):1278–1296, 1987.

[6] A. Bertolino. An overview of automated software testing. *J. Syst. Softw.*, 15(2):133–138, 1991.

[7] E. Börjesson and R. Feldt. Automated system testing using visual GUI testing tools: A comparative study in industry. In G. Antoniol, A. Bertolino, and Y. Labiche, editors, *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, pages 350–359. IEEE Computer Society, 2012.

[8] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *J. Syst. Softw.*, 152:165–181, 2019.

[9] M. Caulo and G. Scanniello. A taxonomy of metrics for software fault prediction. In *46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020, Portoroz, Slovenia, August 26-28, 2020*, pages 429–436. IEEE, 2020.

[10] K. S. M. Chan, J. Bishop, J. Steyn, L. Baresi, and S. Guinea. A fault taxonomy for web service composition. In E. D. Nitto and M. Ripeanu, editors, *Service-Oriented Computing - ICSOC 2007 Workshops, International Workshops, Vienna, Austria, September 17, 2007, Revised Selected Papers*, volume 4907 of *Lecture Notes in Computer Science*, pages 363–375. Springer, 2007.

[11] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M. Wong. Orthogonal defect classification - A concept for in-process measurements. *IEEE Trans. Software Eng.*,

18(11):943–956, 1992.

[12] A. M. F. V. de Castro, G. Macedo, E. Collins, and A. C. Dias-Neto. Extension of selenium RC tool to perform automated testing with databases in web applications. In H. Zhu, H. Muccini, and Z. Chen, editors, *8th International Workshop on Automation of Software Test, AST 2013, San Francisco, CA, USA, May 18-19, 2013*, pages 125–131. IEEE Computer Society, 2013.

[13] F. Dobslaw, R. Feldt, D. Michaelsson, P. Haar, F. G. de Oliveira Neto, and R. Torkar. Estimating return on investment for GUI test automation tools. *CoRR*, abs/1907.03475, 2019.

[14] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner. Security testing: A survey. *Adv. Comput.*, 101:1–51, 2016.

[15] K. Fögen and H. Lichter. Combinatorial robustness testing with negative test cases. In *19th IEEE International Conference on Software Quality, Reliability and Security, QRS 2019, Sofia, Bulgaria, July 22-26, 2019*, pages 34–45. IEEE, 2019.

[16] B. G. Freimut, C. Denger, and M. Ketterer. An industrial case study of implementing and validating defect classification for process improvement and quality management. In *11th IEEE International Symposium on Software Metrics (METRICS 2005), 19-22 September 2005, Como Italy*, page 19. IEEE Computer Society, 2005.

[17] D. J. Garney. A software structure for testing a complex product with a minicomputer. In A. K. Hawkes, editor, *Proceedings of the 26th ACM annual conference, ACM 1971, USA, 1971*, pages 191–196. ACM, 1971.

[18] D. Gelperin and B. Hetzel. The growth of software testing. *Commun. ACM*, 31(6):687–695, 1988.

[19] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Á. Beszédes, R. Ferenc, and A. Mesbah. Bugsjs: a benchmark of javascript bugs. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pages 90–101. IEEE, 2019.

[20] Q. Hanam, F. S. D. M. Brito, and A. Mesbah. Discovering bug patterns in javascript. In T. Zimmermann, J. Cleland-Huang, and Z. Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 144–156. ACM, 2016.

[21] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.

[22] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella. Taxonomy of real faults in deep learning systems. *CoRR*, abs/1910.11015, 2019.

[23] J. Itkonen and K. Rautiainen. Exploratory testing: a multiple case study. In *2005 International Symposium on Empirical Software Engineering (ISESE 2005), 17-18 November 2005, Noosa Heads, Australia*, pages 84–93. IEEE Computer Society, 2005.

[24] E. Kinsbruner. Manual Testing vs. Automated Testing | by Perforce.

[25] M. Lackovic, D. Talia, R. Tolosana-Calasanz, J. A. Bañares, and O. F. Rana. A taxonomy for the analysis of scientific workflow faults. In *13th IEEE International Conference on Computational Science and Engineering, CSE 2010, Hong Kong, China, December 11-13, 2010*, pages 398–403. IEEE Computer Society, 2010.

[26] D. L. Lough. A taxonomy of computer attacks with applications to wireless networks, 2001.

[27] L. Mariani. A fault taxonomy for component-based software. *Electron. Notes Theor. Comput. Sci.*, 82(6):55–65, 2003.

[28] L. Mariani, M. Pezzè, and D. Zuddas. Recent advances in automatic black-box testing. *Adv. Comput.*, 99:157–193, 2015.

[29] J. D. McGregor. An overview of testing. *J. Object Oriented Program.*, 9(8):5–9, 1997.

[30] A. Nikanjam, M. M. Morovati, F. Khomh, and H. B. Braiek. Faults in deep reinforcement learning programs: A taxonomy and A detection approach. *CoRR*, abs/2101.00135, 2021.

[31] C. Oezbek. Introducing automated regression testing in open source projects. *CoRR*, abs/1001.0683, 2010.

[32] M. Sánchez-Gordón, L. Rijal, and R. C. Palacios. Beyond technical skills in software testing: Automated versus manual testing. In *ICSE '20: 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*, pages 161–164. ACM, 2020.

[33] O. Taipale, J. Kasurinen, K. Karhu, and K. Smolander. Trade-off between automated and manual software testing. *Int. J. Syst. Assur. Eng. Manag.*, 2(2):114–125, 2011.

[34] O. Tal. *Software dependability demonstration for safety-critical military avionics systems by statistical testing*. PhD thesis, Nottingham Trent University, UK, 1999.

[35] K. F. Tómasdóttir, M. F. Aniche, and A. van Deursen. The adoption of javascript linters in practice: A case study on eslint. *IEEE Trans. Software Eng.*, 46(8):863–891, 2020.

[36] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verification Reliab.*, 22(5):297–312, 2012.

[37] M. Vieira, X. Song, G. Matos, S. Storck, R. Tanikella, and W. M. Hasling. Applying model-based testing to healthcare products: preliminary experiences. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 669–672. ACM, 2008.

[38] E. J. Weyuker and F. I. Vokolos. Experience with performance testing of software systems: Issues, an approach, and case study. *IEEE Trans. Software Eng.*, 26(12):1147–1156, 2000.

[39] M. Young and R. N. Taylor. Rethinking the taxonomy of fault detection techniques. In L. E. Druffel, D. Fairley, and D. Bjørner, editors, *Proceedings of the 11th International Conference on Software Engineering, Pittsburg, PA, USA, May 15-18, 1989*, pages 53–62. IEEE Computer Society / ACM Press, 1989.

Towards Aspects of Cloud Computing

Ricky Jonathan
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
ricky.jonathan@rwth-aachen.de

Joana Schmidt
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
joana.schmidt@rwth-aachen.de

ABSTRACT

Cloud computing is a concept that allows users to utilize – instead of their own computing resources – on-demand computing resources offered by cloud provider through the internet. This has been widely used because of its potential to improve business operation through cost efficiency, easy maintenance, high degree of scalability, etc. However, lots of aspects or terms in the context of cloud computing remain unclustered and are still not well-defined. This may lead to difficulty to understand the whole concepts in and of cloud computing. Moreover, this leads to other difficulty to identify and tackle some problems related to cloud services, as well as to measure the Quality of Service (QoS) guarantee of cloud services accurately. This paper contributes to create an approach that abstracts the aspects of cloud computing and elaborate the importance of those aspects. In this way, cloud computing aspects can be generalized into the basic *must-have* cloud service requirements and would be easier to comprehend.

Categories and Subject Descriptors

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

Keywords

Cloud computing, cloud dimension, cloud service requirements

1. INTRODUCTION

1.1 Background

Cloud Computing has been well-known for establishing virtualized on-demand services (e.g., storage, computing service, database) to its users. The appearance of cloud computing was concerned with the limitation by data center to flexibly scale in/out in order to deal with evolving business challenges. The usage of classic data center has a great risk of over-provisioning and therefore causes additional cost by companies. Furthermore, companies using server on data center also need to hire IT staff to manage and secure the IT infrastructure. As cloud computing usage is based on pay-per-use service and provides virtualized IT infrastructure, it has been preferred nowadays to reach economies of scale. As depicted in Figure 1, cost efficiency has become the reason for the use of cloud services. For instance, cloud storage (e.g., Dropbox) is one of the biggest advantages provided by cloud service provider.

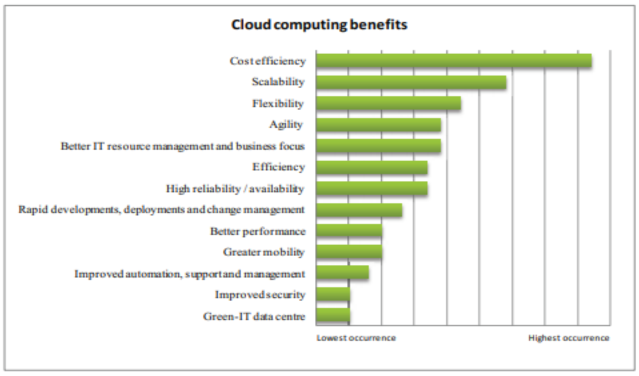


Figure 1: Cloud computing benefits [6]

1.1.1 *Characteristic of Cloud Computing*

In order to have better understanding about cloud computing, we specify the essential characteristics of cloud computing introduced by NIST (National Institute of Standard and Technology) [14]: On-Demand self-service, broad network access, resource pooling, rapid elasticity, and measured service. **On-Demand self-service**: Cloud service provisioning is highly automated and customers do not need human interaction with each service provider. **Broad network access**: Service are reachable over the internet. **Resource pooling**: Computing resources are pooled to serve multiple consumers. **Rapid elasticity**: Services can be elastically provisioned to scale according to demand. **Measured service**: Cloud systems automatically control and optimize resource use by leveraging a metering capability.

1.1.2 *Deployment Models of Cloud*

Furthermore, depending on the relationship among cloud provider and users, the deployment models of cloud comprise of public cloud, private cloud, community cloud, and hybrid cloud [14] [23]. **Public cloud**: The term public refers to open service for public. Thus, universities, companies, or government may benefit from the usage of public cloud. **Private cloud**: This type of cloud allows only exclusive use of the infrastructure to a certain organization. This type of cloud is reasoned by privacy issues. **Community cloud**: The use of the service is only allowed for specific communities that have the same concerns. For example, business organizations that working together on the same business goals may use community cloud. **Hybrid cloud**: This is the mixture of two or more cloud service types (public, private, or community). This can be useful, as an organization is able to keep privacy of sensitive data on private cloud, while application with less compliance requirements could be provided via public cloud to reach economies of scale.

1.1.3 Service Models of Cloud

NIST [14] also categorizes cloud computing into 3 service delivery types such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

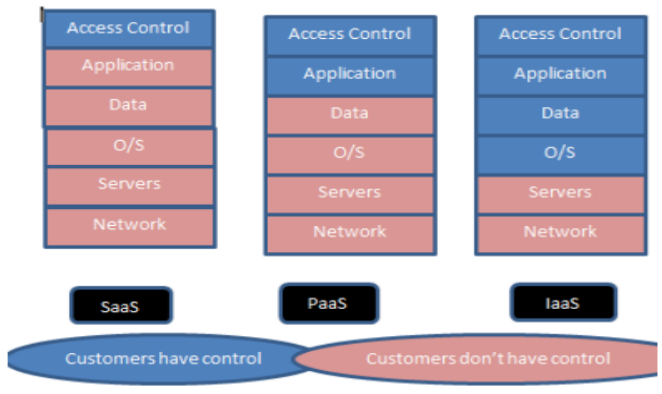


Figure 2: Customers' control over different layers in different service models [8]

IaaS: a cloud service provider (CSP) delivers IT infrastructure including server, network, storage, and operating system. Cloud servers are delivered through virtualization via API, granting access for the clients to the infrastructure. However, users need to maintain hardware and software on their own, including the operating system, runtime, application, security, database, etc. Hence, IaaS is regarded as self-service business model in cloud computing. Companies that are ready to invest the time on maintaining the abovementioned aspects can benefit from IaaS, as IaaS is known by its high degree of flexibility (independent of platform) and it also allows clients to have complete control of the infrastructure, so that clients can specify them based on their needs. The other advantages are that clients, especially startup companies, could avoid spending money on buying the ready software product. In the other hand, the downside is additional work for managing the infrastructure. IaaS clients need to train the staff to manage all the resources, as they are responsible for managing the infrastructure. The most prominent example of IaaS is Amazon Elastic Compute Cloud (Amazon EC2), introduced in 2006.

PaaS: a CSP not only delivers, but it also manages cloud environment including server, storage, network, operating system, runtime, database, and security. In other words, PaaS gives the ready-to-use platform for the clients to create an application. In this way PaaS enables users to customize their own software application without caring about the underlying infrastructure. The other advantages is that PaaS also provides services to test and deploy the customized application on its platform. However, PaaS has less flexibility than IaaS in terms of scale in/out because PaaS has no control over the underlying infrastructure. Microsoft Azure and Google App Engine are examples of big PaaS-providers.

SaaS: a CSP provides complete software product where services are hosted from remote server. Users can benefit from this type of service, as they do not need to spend time and effort to maintain software update, as it is managed by the provider. However, the drawbacks are lack of control over data, low degree of software customization, and possible downtime due to, for example, system failure or software upgrades. Typical example of SaaS is Google Apps.

There exists more advantages and disadvantages which emerge generally as characteristics of cloud computing services such as high efficiency, measurable service delivery, pay-as-you-go services, security issues, possibility of data loss, vendor lock-in, etc. In this era, cloud service has already been used all over the world, but concerns about privacy of data stored in cloud are growing rapidly and need to be addressed accordingly.

1.2 Motivation

Cloud computing has wide range of functionality and therefore there exists lots of aspects that need to be covered. For example, an end-user can concern the capacity storage and data confidentiality of a cloud service. Another example could be a cloud provider who concerns about how efficient the services are automatedly provisioned to the user. As a matter of fact, lots of aspects of cloud computing remain unclustered and this leads to difficulty in measuring a use of cloud computing services. Moreover, this also can be shown by Fehling's dissertation [7], which tells the existence of hundreds of patterns created in the context of cloud computing in order to tailor the type of cloud services according to each customers' needs. This means that there may exist extremely much problem-solution pairs needed to be discovered and it is believed that there may exist even much more out there in the future. This paper contributes to create an approach to elaborate the aspects of cloud computing more thoroughly.

2. RELATED WORK

A cloud provider has responsibility to meet its client request. Therefore, several aspects of cloud computing must be taken into account in order to comply with standard requirements, before proceeding with specific details from clients. Requirements engineering processes are activity that deal with determination, documentation, validation, and management of the requirements. In the paper introduced by Zalasar et al. [24], there exists five dimensions of cloud requirements engineering, consisting of contractual, financial, compliance, operational, and technical dimension.

Contractual dimension deals with the determination of cloud Service Level Agreement (SLA) between cloud provider and client, consisting of agreement that certain services with certain expected level of service delivery must be provided to the client. In general, the most important components of SLA between both parties are including contract duration, purpose of contract, pricing model, document approvals, and responsibilites of actors involved. Furthermore, a cloud SLA should contain not only types of services, but also the quality of service (QoS) that would be provided, such as performance, reliability, availability, cost, etc. There also exists penalties in a complete SLA if the clients' requirements are not met. Moreover, in order to measure the QoS accurately, metrics must be taken into account as key performance indicator (KPI) of a cloud provider [3]. **Financial dimension** refers to problem related to pricing models, accounting and billing. As cloud usage is considered as pay-per-use service, where people pay proportionally to the amount of resource or time they used, the financial aspect is becoming a focus of both providers and clients. **Compliance dimension** deals with legal and security issues by providing and using cloud services. In order to prevent malicious attacks or fraud, cloud provider should be willing to proceed with security protocol, such as by being transparent about the geographical location where the data is stored, and also by proving that the location is formally permitted by governance policies and agreed upon the contract. **Operational dimension** is essential for keep the service running well and always resilient given a sudden change or an unexpected failure by the delivery of service. **Technical dimension** sets functional and non-functional requirements of cloud services. The functional properties are regarded as the service that will be provided, while the non-functional properties are related to the Quality of Service, such as performance, reliability, availability, cost, etc [17]. All these dimensions classify specific requirements of cloud services, that subject to a certain cloud service agreement.

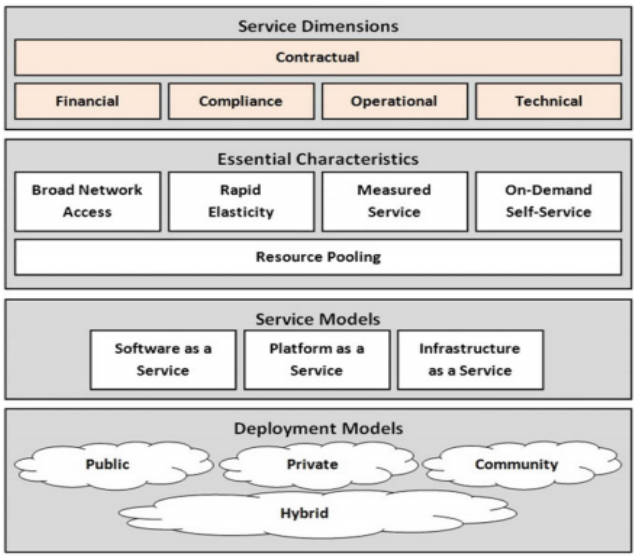


Figure 3: Extension of the NIST cloud definition framework [14]

Another dimension classification of cloud computing is presented by Repschlaeger et al. [16] . In his paper he proposes cloud computing dimension based on user's objective. These are, as shown on Figure 5, consisting of service and cloud management; IT security and compliance; reliability and trustworthiness; scope and performance; costs; and flexibility. Each target dimension represents a general objective which customers pursues and which characterizes his cloud or IT strategy. The service and cloud management provides functions for cloud service operations such as for controlling and monitoring service performance. Then, IT security and compliance guarantee shielding and protection of data and applications in cloud against unauthorized access. Another target dimension - reliability and trustworthiness - ensures availability of services and that the service level corresponds to the conditions of Service Level Agreements (SLAs). Another target dimension scope and performance deals with purpose of service as well as specification of QoS guarantee. The target dimension costs contains pricing and billing models as well as payment methods, which are agreed upon SLA. Flexibility as another target dimension considers scalability aspect and adaptability to changing capacity requirements.

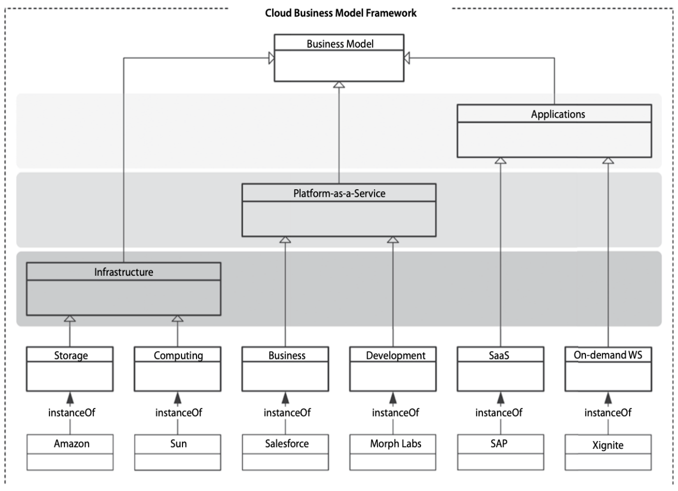


Figure 4: Cloud business model framework [21]

The last model for cloud computing we will point out, is introduced by Weinhardt et al. [21]. In his paper he presents a framework for business models in cloud, which can be categorized in infrastructure, platform and application as shown in Figure 4. This can be regarded as the implementation of what the background section of this paper above have elaborated about IaaS, PaaS, and SaaS.

However, an approach to model a process of offering and acquiring of cloud service has never come into discussion yet. This modelling approach is essential for the readers before getting familiarized with the aspects of cloud computing. The main contributions of the paper is to model a use case diagram specifying activities done by both user and cloud provider. Then, we elaborate how the aspects of cloud computing are covered along with the interactions between user and cloud provider.

3. AN APPROACH TO ABSTRACTION OF CLOUD COMPUTING

We present a use case diagram which shows the interconnection between user and cloud provider. This helps to understand the flow of offering and acquiring a cloud service by cloud provider and user respectively, shown in Figure 6. Then, the table 1 shown below defines the use case actions.

Some companies have discovered positive impacts of cloud

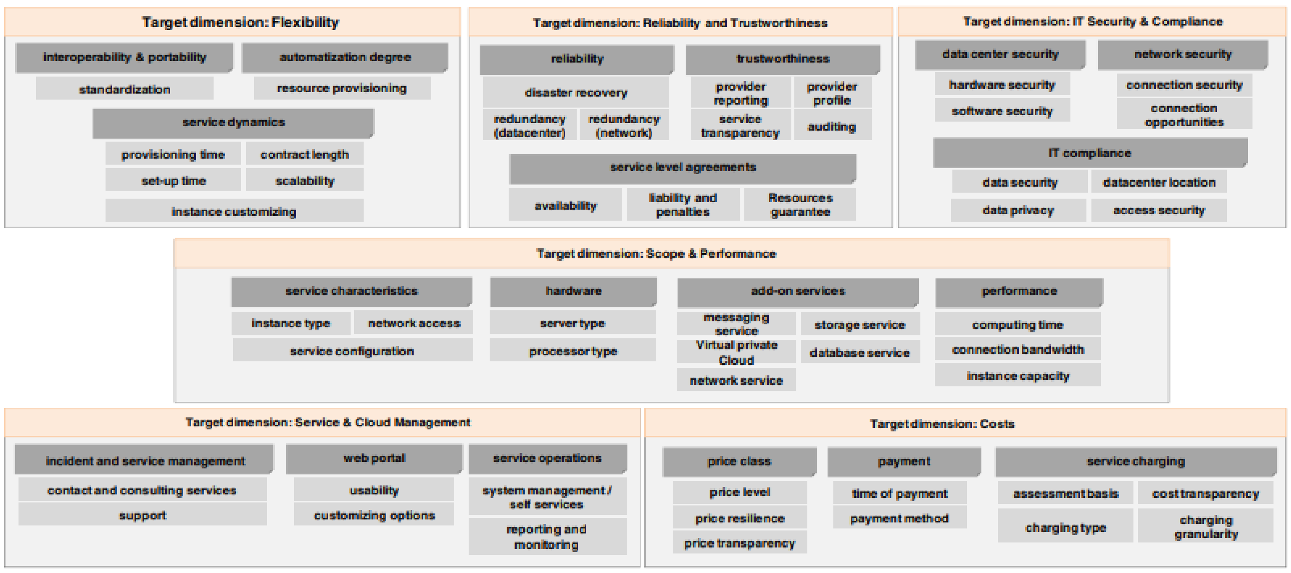


Figure 5: Target dimension of cloud computing [16]

computing on their business processes. Nevertheless, they have to estimate the cost for developing and deploying application in a cloud offered by a provider, including IT infrastructure, data centre equipment, software licenses, software maintenance, etc. In some cases it is even more reasonable to build on premise-infrastructure at *one time payment* rather than to make use of cloud service, e.g., to avoid long-term costs. Cloud computing has several downsides on its own, such as cloud platform interoperability, security issues, and service availability [4]. Hence, enterprises have to make a decision regarding the cloud adoption, whether to buy or lease the IT services, by considering the benefits, risks, and also financial factors.

Moreover, companies, as a user, plan the details of service they are willing to have. The fundamental ingredient for customer to obtain a suitable service is the type of service such as SaaS, PaaS, or IaaS, because any later service requirements (e.g., cost) could not be clearly defined without it. For example, if a project is considered as a short-term project that would not cause big expense, the company should invest such an expense to SaaS provider as the company does not have to take another effort and additional cost for maintaining hardware and software, which is unnecessary for short-term project.

Then, comparing existing cloud providers is essential to get the "product knowledge" offered by them. A key element for comparing those is Service Level Agreement (SLA). This is a service-based agreement, which proposed by cloud provider to all customers using the service. Users having the service requirements read the policy and terms of use and analyse the QoS guarantee stated in Service Level Objective (SLO). Then, they need to know which KPIs a cloud provider uses to measure the QoS, before finally consider if such service requirements are possible to be fulfilled by cloud provider. However, the complex environment of cloud arises the challenge to precise the QoS guarantee and the cause of possible service interruptions. Additionally, Figure 7 [5] simplifies the concept of SLA. QoS is considered as Non-

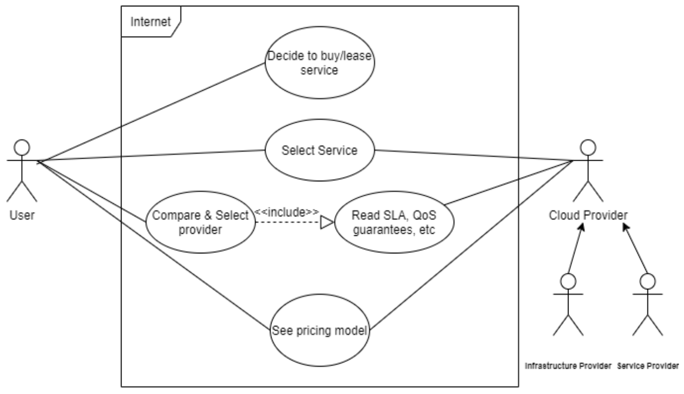


Figure 6: Use-Case Diagram: Offering and Acquiring Cloud Service

Functional Requirements (NFR) and KPI is considered as Non-Functional Properties (NFP).

As shown in Figure 8 [3], both functional and non functional properties reside at each cloud service model. Functional properties are merely about what kind of service a cloud provider give to a user and therefore easy to be defined. For instance, Amazon EC2, as a IaaS cloud provider, provides several instance types (e.g., Mac instances) for users to develop, build, and test a software on those instances. Amazon RDS PaaS cloud allows user to schedule queries, operate, easily scale their relational database, and backing it up in the cloud. As SaaS cloud provider, Google Apps provided Google Drive that allows user to back up, update, and share documents.

Non-functional properties are instruments to measure service delivery performance. These are related to cost efficiency, availability, security, and many other KPIs. Depends on what kind of service a cloud provider delivers (e.g., database, storage, microservices, etc), only some certain amount of KPIs are picked for the performance target.

Table 1: Use Case Definition

Action	Definition
Decide to buy/lease service	User makes cost estimation, benefit, risk, and opportunity cost for developing and deploying application in an offered cloud by provider for a certain period of time. A paper that provides modelling tool to support this decision making of acquiring IaaS clouds is presented by Khajeh-Hosseini et al. [11]. The authors consider technical, financial, and compliance benefits and risks in the development of the tool. This action is often accompanied with comparing different cloud providers and - not limited to - deployment options as well as usage scenarios.
Select service	User considers the best usage scenario based on their actual needs. As stated in the introduction section, there exists 3 main service models of cloud, namely IaaS, PaaS, and SaaS. Cloud provider introduces the service model that it offers.
Compare and select provider	User considers all important aspects, including cost, QoS guarantee, security, legal, etc, and finally select the most suitable provider. Cloud provider presents Service Level Agreement (SLA), which is of main interest for users, as it contains all information regarding agreement and service specification.
See pricing model	Cloud provider creates user-centric pricing model that enables usage optimization of cloud service for reasonable cost. The basic pricing model used in cloud service pay-per-use and subscription-based pricing. User takes pricing model into account. Wang et al. [20] propose a pricing strategy using Markov decision process to maximize the revenue of cloud service providers.

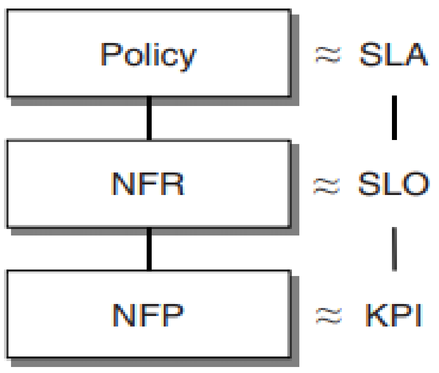


Figure 7: SLA, SLO, KPI [5]

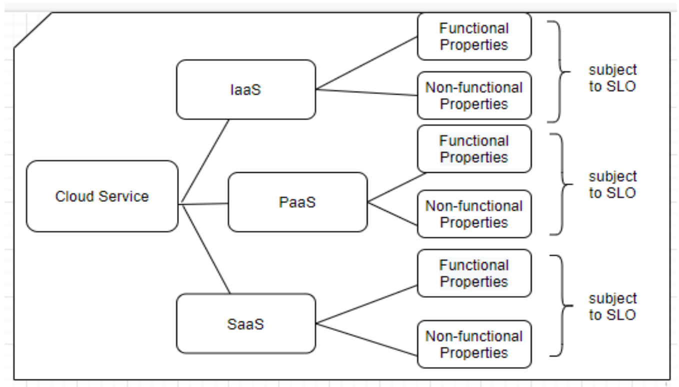


Figure 8: Requirement of cloud service at each service model [3]

This should be clearly stated in cloud SLA to keep the transparency of QoS guarantee for customers.

Finally, pricing model affects customers' decision on choosing cloud service at reasonable cost. In cloud computing, pay-per-use (e.g., Amazon EC2, price per hour) and subscription based pricing (e.g., Adobe Creative Cloud, monthly subscription fee) are the standard choice that cloud providers use [9]. Cloud providers offer not only attractive pricing scheme, but also consider investment costs, maintenance cost, QoS guarantee, as well as supply and demand behaviour on market. In this way, cloud provider may have a beneficial and user-centric service offer. This leads to a fact that costs play a major role by both parties.

3.1 Relation of Use Case to Cloud Computing Dimension

The use case diagram models a process that covers all the aspects mentioned by Zalasar et al. [24]. We present the terms of cloud computing related to those aspects below in the next section.

Financial dimension is covered by users foreseeing the cost efficiency, cost, and risk before acquiring (buy or lease) cloud service. The objective of providers is that to provide users with services that can help them to reach economies of scale. Policies and terms of use, covering the **compliance dimension**, are described in cloud SLA to ensure security protocol in service delivering process in certain period of time agreed upon the contract. Nowadays, cloud providers tend to use different techniques of cryptography such as block cipher, stream cipher, hash function, etc [2] in order to mitigate risk of data leakage. However, there still exists threat that cloud providers still may, due to full control of services, have their security being compromised if they lose their control over data. For example, Amazon give a statement in its cloud SLA that they are not responsible for any data corruption, unauthorized use, or any other related terms that bring harm to the application [12].

Operational dimension takes care of responsibilities of all parties involved in cloud SLA. Its existence to ensure that

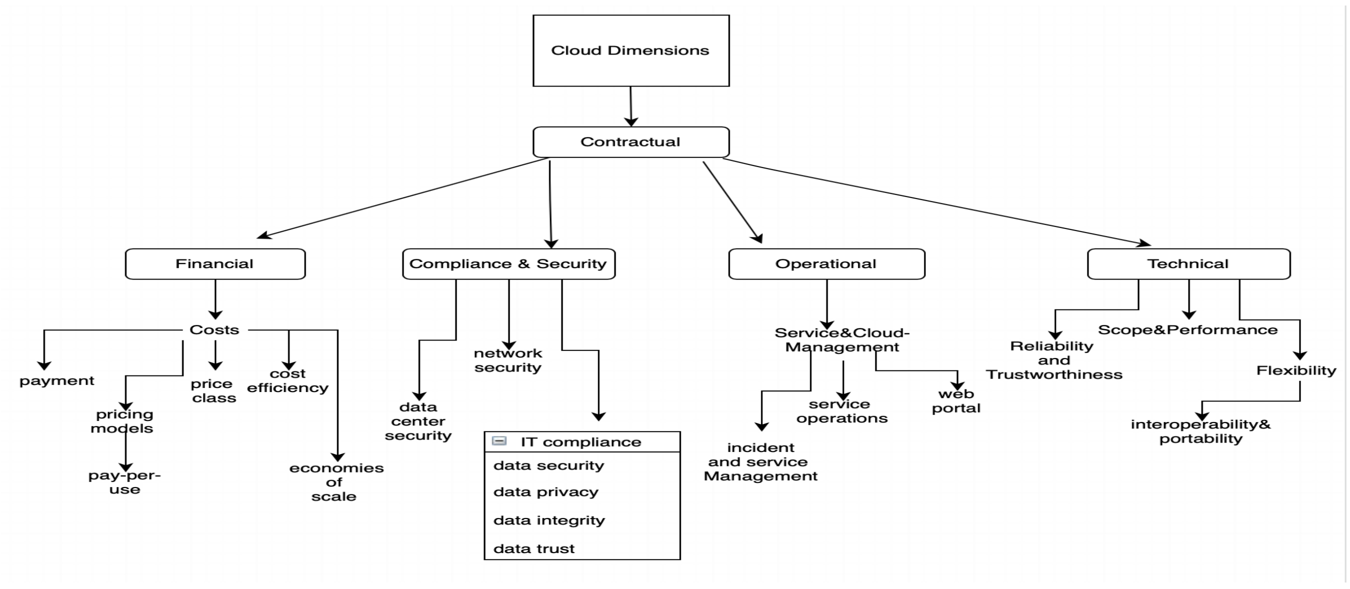


Figure 9: Differentiation of Cloud Computing Dimensions

the process of service delivery runs as agreed upon the contract. This consists of service response, service priority, and service management. The prominent example could be measurement of service availability using Mean-Time-To-Recover (MTTR) or Mean-Time-Between-Failure (MTBF). MTTR measures the time needed by a system after downtime to re-operate according to the service specification, while MTBF specifies the average time of failures of service in a normal operation. Therefore, the less amount of MTTR or MTBF means the better process delivery. Finally, **technical dimension** is covered by users comparing result specification offered by cloud provider, e.g., how a provider chooses metrics to evaluate the target fulfilment. Non-functional properties measured by KPI are the key element to figure out the efficient solution.

Contractual dimension ensures good communication between user and cloud providers. To recap, this dimension is related to all aspect of cloud SLA such as offered service model (IaaS, PaaS, or SaaS), pricing model, service management, service response, availability, metrics, responsibilities of actors, etc. We believe that financial, compliance, operational, and technical dimension are a part of contractual dimension, as all of them are essential for establishing cloud SLA and the fact that contractual dimension is the SLA itself that holds those four dimensions together. Thus, comparing cloud providers by analyzing the content of cloud SLA would also consider all the abovementioned dimension of cloud computing. However, it is important to know that there are a bunch of terms derived from each dimension level.

3.2 Refinement of Cloud Computing Dimension

In Figure 9 we try to generalize the cloud computing terms from each cloud dimension into the most basic *must-have* cloud service requirements. For the base of our sketch, we use the classification of cloud dimensions from Zalasar et al. [24], consisting of contractual, financial, compliance security, operational and technical dimension. All of these dimension were refined by related aspects of cloud computing dimension, which are provided by Repschlaeger et al. [16] [22].

The financial dimension is responsible for financial and economic aspects of cloud computing. Costs play a major role there, because cloud computing offers monetary advantages, for example small capital commitment or that the costs for the resources required are lower. However, providers of cloud services use different pricing and -billing models. Pay-per-use can be cited as an example of a pricing model, with this method the customer has to pay for the services used by him either fully or partially. Another aspect of the costs is the payment, which includes the payment method consisting of credit card or bank transfer and also the time of payment, i.e. whether the costs for the services are debited in advance or afterwards. Price class is another criterion that influences the costs. This includes the price level, price resilience and price transparency, i.e. which information about price options or resilience are available. A strong benefit of cloud computing is the cost efficiency that also contributes to the costs in the financial dimension. Therefore, cloud computing services offer cost advantages, for example the reduction of investment and operating costs. Besides, economies of scale are another significant advantage of cloud computing, because of that larger providers have lower costs for delivering a service. Moreover, cloud provider can deliver better services due to the more reliable infrastructure and availability of resources. Furthermore, they provide deeper expertise for customers and also make use of Green IT, i.e. combined operations that can reduce power consumption [13].

The compliance security dimension is responsible for legal and regulatory aspects of cloud services and their restrictions. Data and applications that are utilized in the cloud need to correspond to compliance guidelines and be secured against unauthorized access. IT-compliance is one aspect of this dimension that includes data security, data privacy, data integrity and data trust. The main goal in data security is the protection of sensitive data. To ensure this, different technique for example Encryption technique were often used to secure outsourced data. Data privacy is one security challenge of cloud computing. It can be assured by the cloud services preventing opponents from inferring the user's behavior, when they visit the sensitive data, i.e. it is not a direct data loss. Data integrity guarantees the customer reliability and uniformity of data, so that the data of the customer can not be changed or manipulated without the customer's approval. If somebody has unauthorized access to the stored data of the customer, then the data integrity is at risk. Therefore, it is important to insure data integrity for avoiding data corruption and data crash. Data trust is also an essential aspect between the customers and cloud service providers. Because the technology and control of the data is non-transparent for the customers, trust is only based on the security which the provider conveys to the customers. In order to build trust in the relation between customer and provider, reputation, isolating data without contravening integrity and the privacy issues as well as transparency of data are important measures [18]. Another aspect of this dimension is the data center security which is not related to the customer but only to the provider. The provided security of the data center includes for example building protection or virus protection. Furthermore, network security can be grasped as a criterion that describes the security of the provided infrastructure and also the communication protection.

The operational dimension is responsible for ensuring that the cloud services run according to the agreed contract. The main components of it are service response, service priority, and service management. The service cloud management is authoritative for appropriate cloud service operations by providing corresponding features of the provider, the main focus is on the requirements and responsibilities by the customer. It consists for example of offered support for access control and of the possibility to design the web interface individually. Incident service management is one part of service cloud management that includes appropriate support under certain circumstances as well as customer service. Furthermore, service operations are essential for the service cloud management, because they control and manage cloud services by offering monitoring and- reporting functionalities. In addition, their existence is necessary to update and replace software and hardware and also for maintenance. Web portal as another criterion of service cloud management deals with the usability of the surface of the web portal and also with functions for the customer to adapt this surface by providing customizing options.

The technical dimension is responsible for functional and measurable attributes of cloud service. It can be divided into functional and non-functional requirements of cloud services, where functional properties are related to the service that will be provided and non-functional properties to the Quality of Service. One basic requirement includes scope performance, i.e. the scope of services and the performance of a cloud provider. It covers aspects of the hardware, so which processor type or server type is used as well as aspects of service characteristics like available operating systems, network access or service configuration. Furthermore, add-on services are part of the scope of services for instance storage service, database service or messaging service which are all additionally bookable. The criterion performance is related to the maximum limit of instance capacity and to the required computing time as well as connection bandwidth, i.e. transfer volumes and transfer speed. Another non-functional requirement describes the flexibility of cloud computing services, which implies adaptability to changing capacity requirements.

Consequently, resources are allocated and released as required and the provisioning time is reduced. Moreover, interoperability and portability is a significant feature of flexibility, because nowadays, there is a wide variety of digital media which are distributed over different networks. Therefore, a standardization is important to guarantee interoperability between clouds as well as transcoding of media contents [1]. On this account media cloud enables the customer to manage media content transparently, even across users domains and Inter-cloud computing, i.e. communication between two or more clouds, is required to identify services and to create more services in order for better provisioning of services in dealing with multimedia [1]. To meet the different requirements of the users, one cloud needs to interact with another cloud or multiple clouds, what creates 'Cloud of Clouds' (CoC) allowing direct data communication [1]. The prerequisites required for this are the standardization of cloud interoperability whereby service level agreement (SLA) has to be integrated there as well as an Inter-cloud Protocol [1]. Portability of cloud services is also of great relevance in cloud computing, because it allows the users to transfer their data and application from one platform to another [10]. This can be the case, e.g. when the user wants to change from private cloud to hybrid cloud [10]. The migration of the data to another platform and networked work in the cloud is required to ensure portability [10]. The last non-functional requirement of technical dimension will be reliability and trustworthiness, which guarantee availability and operational readiness of cloud services according to the Service Level Agreements (SLAs). In detail, reliability refers to the compliance with provider's service commitments and is ensured by disaster recovery plan, redundancy system that means different data centers and network connections from different internet service providers as well as safety measures for personal data. The criterion of trustworthiness includes service transparency as well as the provider profile and provider reporting which is gained through his business activities, experience in the market and auditing.

In summary, it can be said that this refinement of cloud computing aspects is not the only possibility. Because the limits of the dimensions are fluid, which is why a clear assignment of an aspect to a dimension does not have to be unambiguous.

4. DISCUSSION

Although the identified use case diagram can be used to model the interaction between cloud user and provider, this still has drawbacks regarding the abstraction of cloud computing aspects. It contains only, as a use case diagram in general, the actors involved with specific roles and the actions performed by actors. However, there exists many possible scenarios which can not be described by a single use case diagram. For instance, this does not explain what would happen while using the cloud service if a user realize to have miscalculated its own optimal target requirements and consider that the service would not suffice to fulfill the revised requirements. A possible scenario would be to switch a new contract and to pay the penalties for breaking a duration contract, or to *re-analyze* and to maintain the requirements so that the cloud service currently used could at least give a certain value to the user by the end of the contract, although the optimal requirements are not met at all. This creates a hole in describing the whole cloud computing aspects. In addition, the use case diagram – unlike an activity diagram – can not create a workflow that put the actions into the sequences that could be more comprehensive for the readers.

We also believe that the refinement and classification of cloud computing aspects is not simple because of the fact that each cloud user has its own targets and ways to consider the requirements for an optimal delivery of service. Figure 9 is the generalization of the whole cloud computing aspects that restricts the terms of cloud computing into the basic *must-have* needs in the cloud service.

However, many of these cloud computing aspects are exposed to dangers and face major challenges. Figure 10 shows a list of the identified risks of cloud computing based on literature research [6]. It includes the risks of security, third party vendors, management and control, laws and regulation, portability and interoperability, disaster recovery, virtualization risks, lack of standards and auditing, maturity of technology and uncontrolled viable costs. The diagram shows that security is the biggest cloud computing risk. The user saves all of his or her sensitive data in the cloud, consequently the cloud provider has to ensure that the personal data is secured against use by third parties. One security threat is data breach, i.e. when sensitive data is stolen by unauthorized access or even by insider person. For instance, a leak of financial information can damage large companies. Furthermore, data loss is another security threat, because the personal data as well as the data of the companies which are stored in the cloud, can get lost for example by accidental deleting. In addition, the data can also be threatened by employees by manipulating or leaking the data. The data location also poses a further risk of security, because of that the data should be stored in different location instead of only in one place. Another risk is account hijacking which means that hijackers get unauthorized access to credentials in order to provide and manipulate sensitive data as well as to enter in other's transaction, because they can access all of user's services. Multi tenancy can be listed as a final risk of security, i.e. users share many resources which can cause interference as well as a riskier change management.

In figure 10 [6], it also can be shown that portability and interoperability represent a great risk of cloud computing. Interoperability allows the user to run an application on different platforms and portability guarantees that the users can transfer their data and application from one platform to another [10] [15]. One challenge of Inter-Cloud Computing are „Heterogeneous Media Contents and Media Transcoding“ [1] due to different services that are offered. „Data/Media Sanitization“ [1] represents another challenge of interoperability, because some service providers do not allow customers to save certain data in the cloud. Therefore, the data must pass through a filter and be selected accordingly. Furthermore, „Heterogeneous Media Storage Technologies" [1] can also be seen as a challenge, since multimedia content consume a lot of storage space, it is essential to develop procedures that increase the efficiency to store and search data. In addition, if providers do not use the same media storage technologies, it can result in a reduction in efficiency when clouds communicate. Another risk of cloud computing as shown in figure 10 is disaster recovery which reduce enormous data loss by recovering information from a backup server [19]. There are some challenges such as data storage due to the increased amount of data as well as failure detection which should be able to identify failures quickly to ensure a faster resolution so that the system downtime is reduced. Another challenge of disaster recovery is reliability, which can be ensured by storing the data at several locations. Moreover, replication latency can be seen as an additional problem, because disaster recovery mechanism are dependent on backup replication techniques. There are two different types of these techniques that can be divided into synchronous and asynchronous. Synchronous backup replication has high costs and can affect system performance, whereas asynchronous replication decrease the quality of disaster recovery service. Furthermore, security poses another challenge, because disaster recovery wants to protect data from disasters [19]. Although cloud computing has many advantages, it can be seen that there are also many challenges that cloud computing has to face. Therefore, it is essential to pursue objectives that will support the security of data and minimize risks of cloud computing.

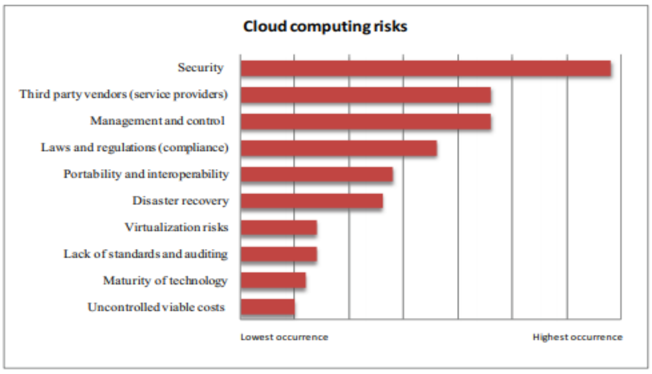


Figure 10: Cloud computing risks [6]

5. CONCLUSION AND FUTURE WORK

Cloud computing aspects are the key for efficient usage of a cloud service. Those are stated on Service Level Agreement (SLA) to ensure the quality of the offered service. However, the QoS guarantee is not always easy to measure by cloud customers under some circumstances, e.g., customers do not even know which aspects should be considered – this means that they are not sure if the offered service level will meet their requirement – , or they are not aware of the KPI chosen by the provider. In cloud computing, only customers with clear definition of SLA metrics can deal with the process of obtaining a suitable service at a reasonable cost. In this paper, the interconnection between cloud user and provider in the form of a use case diagram are presented. Then, the important dimensions of cloud computing were mapped to actions in the use case diagram. In this way, the readers should have an idea which and why cloud computing aspects in each dimension are essential. Finally, the computing aspects were restricted into the most basic *must-have* aspects and established in Figure 9.

As future work, other techniques of abstracting cloud computing aspects can be done in order to elaborate the aspects from different point of view. For example, instead of a use case diagram, an activity diagram is also suitable to abstract the process of the offering and acquiring the service. Activity diagram could describe the sequence of actions that performed in the use case diagram above, so that the workflow of cloud service is easier to understand at a first glance.

Moreover, an extension of cloud computing dimension is still open for the research. The more dimension classification exists, the easier cloud computing aspects can be mapped. Another work could be the mapping of cloud computing aspects to cloud computing design patterns founded by hyperscalers like Google, Amazon, or Microsoft, which tailor its types of cloud services according to its customers' needs, from private organization to public enterprises. In doing so, the result of the work is about establishing the aspect classification of cloud computing design patterns.

6. ACKNOWLEDGMENTS

We would like to thank Alex Sabau, as our supervisor, for the time and effort given for reviewing and giving insights to improve our paper.

7. REFERENCES

[1] M. Aazam and E.-N. Huh. Inter-cloud architecture and media cloud storage design considerations. *the proceedings of 7th IEEE CLOUD, Anchorage, Alaska, USA*, 27, 2014.

[2] A. Albugmi, M. O. Alassafi, R. Walters, and G. Wills. Data security in cloud computing. In *2016 Fifth international conference on future generation communication technologies (FGCT)*, pages 55–59. IEEE, 2016.

[3] M. Alhamad, T. Dillon, and E. Chang. Conceptual sla framework for cloud computing. In *4th IEEE International Conference on Digital Ecosystems and Technologies*, pages 606–610. IEEE, 2010.

[4] S. Bibi, D. Katsaros, and P. Bozanis. Application development: Fly to the clouds or stay in-house? In *2010 19th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises*, pages 60–65. IEEE, 2010.

[5] L. Braubach, K. Jander, and A. Pokahr. A middleware for managing non-functional requirements in cloud paas. In *2014 International Conference on Cloud and Autonomic Computing*, pages 83–92. IEEE, 2014.

[6] M. Carroll, A. Van Der Merwe, and P. Kotze. Secure cloud computing: Benefits, risks and controls. In *2011 Information Security for South Africa*, pages 1–9. IEEE, 2011.

[7] C. Fehling, F. Leymann, R. Mietzner, and W. Schupeck. A collection of patterns for cloud types, cloud service models, and cloud-based application architectures. *University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, University of Stuttgart, Institute of Architecture of Application Systems, Technical Report Computer Science*, 5:19, 2011.

[8] D. Freet, R. Agrawal, S. John, and J. J. Walker. Cloud forensics challenges from a service model standpoint: Iaas, paas and saas. In *Proceedings of the 7th International Conference on Management of computational and collective intElligence in Digital EcoSystems*, pages 148–155, 2015.

[9] R. L. Grossman. The case for cloud computing. *IT professional*, 11(2):23–27, 2009.

[10] H. Gupta and D. Kumar. Security threats in cloud computing. In *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*, pages 1158–1162. IEEE, 2019.

[11] A. Khajeh-Hosseini, I. Sommerville, J. Bogaerts, and P. Teregowda. Decision support tools for cloud migration in the enterprise. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 541–548. IEEE, 2011.

[12] T. A. Lipinski. Click here to cloud: End user issues in cloud computing terms of service agreements. In *International Symposium on Information Management in a Changing World*, pages 92–111. Springer, 2013.

[13] M. Malathi. Cloud computing concepts. In *2011 3rd International Conference on Electronics Computer Technology*, volume 6, pages 236–239. IEEE, 2011.

[14] P. Mell, T. Grance, et al. The nist definition of cloud computing. 2011.

[15] M. Nicho, M. Hendy, et al. Dimensions of security threats in cloud computing: A case study. *Review of Business Information Systems (RBIS)*, 17(4):159–170, 2013.

[16] J. Repschlaeger, S. Wind, R. Zarnekow, and K. Turowski. A reference guide to cloud computing dimensions: infrastructure as a service classification framework. In *2012 45th Hawaii International Conference on System Sciences*, pages 2178–2188. IEEE, 2012.

[17] D. Serrano, S. Bouchenak, Y. Kouki, F. A. de Oliveira Jr, T. Ledoux, J. Lejeune, J. Sopena, L. Arantes, and P. Sens. Sla guarantees for cloud services. *Future Generation Computer Systems*, 54:233–246, 2016.

[18] P. Sirohi and A. Agarwal. Cloud computing data storage security framework relating to data integrity, privacy and trust. In *2015 1st international conference on next generation computing technologies (NGCT)*, pages 115–118. IEEE, 2015.

[19] A. A. Tamimi, R. Dawood, and L. Sadaqa. Disaster recovery techniques in cloud computing. In *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*, pages 845–850. IEEE, 2019.

[20] W. Wang, B. Liang, and B. Li. Revenue maximization with dynamic auctions in iaas cloud markets. In *2013 IEEE/ACM 21st International Symposium on Quality of Service (IWQoS)*, pages 1–6. IEEE, 2013.

[21] C. Weinhardt, A. Anandasivam, B. Blau, N. Borissov, T. Meinl, W. Michalk, and J. Stößer. Cloud-computing. *Wirtschaftsinformatik*, 51(5):453–462, 2009.

[22] S. Wind, K. Turowski, J. Repschlager, and R. Zarnekow. Target dimensions of cloud computing. In *2011 IEEE 13th Conference on Commerce and Enterprise Computing*, pages 231–235. IEEE, 2011.

[23] H. Yang and M. Tate. Where are we at with cloud

computing?: a descriptive literature review. 2009.

[24] A. S. Zalazar, L. Ballejos, and S. Rodriguez. Analyzing requirements engineering for cloud computing. *Requirements Engineering for Service and Cloud Computing*, pages 45–64, 2017.

Carbon Footprint Assessment of Compute Cluster Workloads

Yannick Kahlert
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
yannick.kahlert@rwth-aachen.de

Philip Niederprüm
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
philip.niederpruem@rwth-aachen.de

ABSTRACT

The rising popularity of cloud computing over the past decade has led to an increasing concern about its contribution to global carbon emissions. Our goal is to conduct a literature review that provides an overview of the topic of assessing the carbon footprint of workloads, e.g., web services or data processing jobs running in compute clusters. In the first part of our paper, we clarify how to estimate the carbon footprint of a data center based on four factors, primarily energy consumption.

In section 4, we will proceed with the assessment of workloads. Since a server consumes a certain fixed amount of energy regardless of the utilization, higher utilization rates have a higher energy efficiency. We will see that this correlation between utilization and energy is linear only under certain circumstances and not for all hardware components. We will further present a formula approach to determine the carbon footprint using the PowerTOP software.

In the third and final part, we answer the question of how the carbon footprint of data center workloads can be optimized. To this end, we provide an overview of known solutions and current research approaches. Examples include optimization of the cooling systems or reducing the amount of power consumed in idle states as well as the shifting of workloads to other data centres where their carbon footprint would be lower.

Categories and Subject Descriptors

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

Keywords

carbon footprint, cluster computing, energy consumption, energy efficiency, energy optimization, idle, workload

1. INTRODUCTION

Global data traffic continues to increase, a trend accelerated by the Corona pandemic[11]. While 51% of the world's population had access to the Internet in 2018, that figure will be 66% by 2023 and the number of devices connected to IP networks will be more than three times the global population by 2023 [8]. Data centers, as the factories of the digital age, accounted for 21% of the IT sector's energy consumption in 2017 [9, p. 15] and in total, global IT industry produces as much CO2 as all air traffic or approximately 7% of global electricity [26, 9].

"Across the tech sector we need to recognize that data centers will rank by the middle of the next decade among the large users of electrical power on the planet." [9]

—Brad Smith, President Microsoft

What factors influence the carbon footprint assessment of compute cluster workloads? To answer this question we cannot avoid to look at the data centers on which these workloads are executed. The fast-growing industry has a relevant share in global energy consumption and hence also in CO2 emissions and climate change.

There are many different aspects that influence the environmental impact of a data center. The carbon footprint is highly dependent on the energy mix that the data center draws from. Even if there are some countries with a very high share of renewable energy, the energy price or increased latency due to increased distance to the customer can be reasons not to build data centers in these locations. On the other hand, there are companies that are committed to becoming more environmentally friendly, which, if they do it by their own conviction or not, is good marketing in this day and age.

Consequently, as long as generating energy emits CO2, energy consumption plays a major role in making data centers greener. In addition, there is also an economic interest in reducing energy costs. To this end, we will first explain what the major power consumers of a data center are. In this regard, there have been many efforts and numerous approaches to optimization in research for many years, which we will only touch on in this paper. Furthermore, there are also indirect environment-related factors, such as the disposal and recycling of so-called e-waste, an appropriate usage time of components or the production as well as logistics of spare parts.

How can the carbon footprint of a compute cluster workload be determined or estimated? To this end, we will present a procedure to get to a formula that answers this question. We will further show how the energy consumption of various computing workloads differs before highlighting different techniques to increase data center efficiency. We break these optimization strategies down into software, hardware (since an overall increased efficiency increases the efficiency of a specific workload as well) and further optimization strategies.

2. RELATED WORK

The section that covers the factors contributing to the carbon footprint of data centers is based on a series of reports focusing on the development of carbon intensity over the years made by the International Energy Agency (IEA) [2], the Federal Environment Agency of Germany [1] and the U.S. Energy Information Administration (EIA) [4] respectively. An article from Greenpeace [9] also proved to be important to our research despite usually having rather low credibility, as it contained important information regarding differences in company power policies that was deemed trustworthy enough to be used.

The section on the power consumption of data centers was done using a number of small references such as Koomey's own blog [5], papers by Thurrott [24] and Danilak [10] as well as the aforementioned article by Greenpeace. Another report by the Federal Environment Agency of Germany was also used here. [26]

For the workload discussion we used Meisner's paper on PowerNap [17] with additional information and graphics provided in Tsirogiannis article [25].

When it comes to the topic of optimization, there were numerous papers discussing this. In the end, we decided to focus on Kaushik's and Meisner's papers on GreenHDFS [13] and PowerNap [17] respectively as well as a survey on power saving strategies by Singh. [21]. We chose them because of their clear structure and high density of useful information which almost every paragraph contained. We also reused previous sources such as the report by Borah [7] since they also contained important information about this topic. However, this particular report was rather short, gave only quick overviews and additionally, it suffered from a bad writing style which made it easy to misinterpret points.

3. ENVIRONMENTAL IMPACT OF CLOUD COMPUTING

While other influences exist, the majority of the emissions data centers produce can be tracked back to their power consumption and the associated carbon intensity.

The main concern and primary topic of this paper is the energy consumption of a data center. Unlike carbon intensity, a center's operating company has direct influence on the power distribution and -consumption of their data centers and the implementation of power saving strategies. We'll touch more on the distribution of power and the ways of reducing usage in later sections (section 3.1, section 5). For carbon intensity, in 2019 the International Energy Agency (IEA) reported a carbon intensity of 475g/kWh [2], however, using this to calculate the carbon footprint would lead to inaccurate results as it's the global average and doesn't take regional differences into account. It assumes a homogeneous green/brown (low emission/high emission) power ratio where as in reality this ratio changes on a much smaller scale. For example, German authorities estimated a carbon intensity of 401 g/kWh in 2019 [1] as one can see in figure 1, where as the US reported a value of 417 g/kWh [4]. This fluctuation of carbon intensity extends even further down, with noticeable regional differences. A third factor for calculation is the operator of the data center: In a survey conducted by Greenpeace, noticeable differences in the energy mix of each service provider were found. Each data center receives its energy from different sources which may differ in carbon intensity. In addition, some companies made deals with government institutions to gain access to renewable energy in places where it wouldn't be available otherwise, reducing the emissions caused by the concerned data centers. [9] In conclusion, both a data center's power consumption as well as the carbon intensity of its location and its owner's power policy is needed in order to provide an accurate picture of a data center's carbon footprint.

3.1 Power Consumption of Data Centers

While the well-known "Moore's law" shows a trend that the performance of computers doubles about every 18 months, Koomey found out in 2010 that the energy efficiency (computations per kWh) also doubled about every 18 months (1.57 years) since the very first computers [20, 14]. However, this is no guarantee for the future, especially since one approaches certain physical limits if not already reached (Dennard scaling, second law of thermodynamics, Landauer's principle). Therefore, from the turn of the millennium onwards, the doubling of efficiency slowed down to every 2.6 years [5]. Even though the trend has increased again to 1.2 years since 2014, due to the physical boundaries it is assumed that "Koomey's law" can no longer be observed from 2048 at the latest. [24]

In the long term, efficiency will therefore increase only slowly, whereas the demand for data centers continues to rise, resulting in an even greater energy consumption. While global data center electricity consumption was estimated at about 382 TWh in 2012 [9], in 2016 it was about 416 TWh, almost 40% more than the entire UK. The US share of total consumption in 2016 was 90 TWh [10]. As can be seen in figure 2, even the best case scenario has projected energy consumption more than tripling from 2020 to 2030.

Data centers consume an enormous amount of energy, even though the actual computing consumes less than half of the energy as can be seen in figure 3. Data centers require constant cooling to function properly, which is achieved by installing air conditioning, or by indirect cooling, such as using outside air, indirect evaporative cooling (IDEC) units, and also using sea water.

85% of IT decision makers do not know the energy requirements of their data center [26]. This lack of awareness is surprising, as one of the largest operational expenses in delivering IT is the cost of energy. It suggests that many companies put energy consumption on the back burner and value performance, reliability and stability higher.

Measurement

Many indicators have already been proposed to compare the efficiency of a data center. Widely used is the PUE (power usage effectiveness).

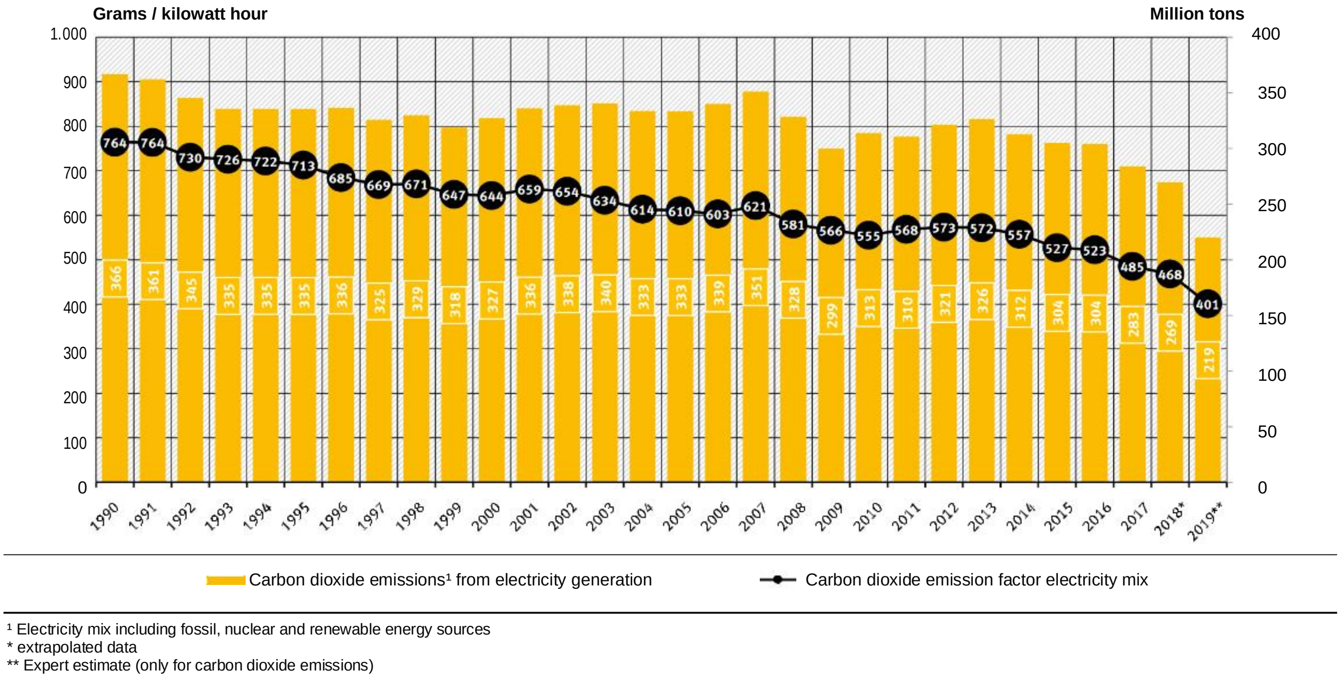


Figure 1: Development of the specific carbon dioxide emissions of the German electricity mix [3]

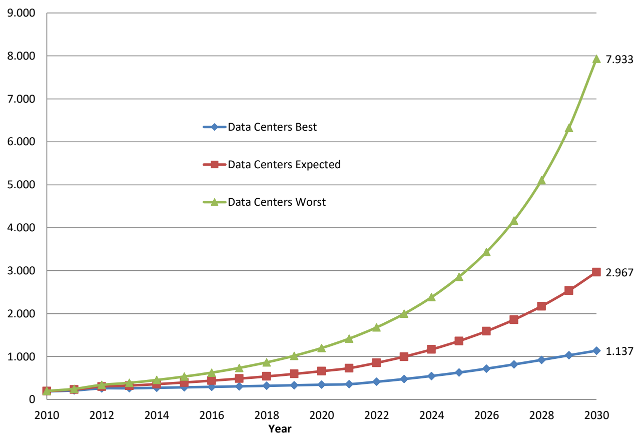


Figure 2: Electricity usage (TWh) of Data Centers worldwide 2010-2030 [6]

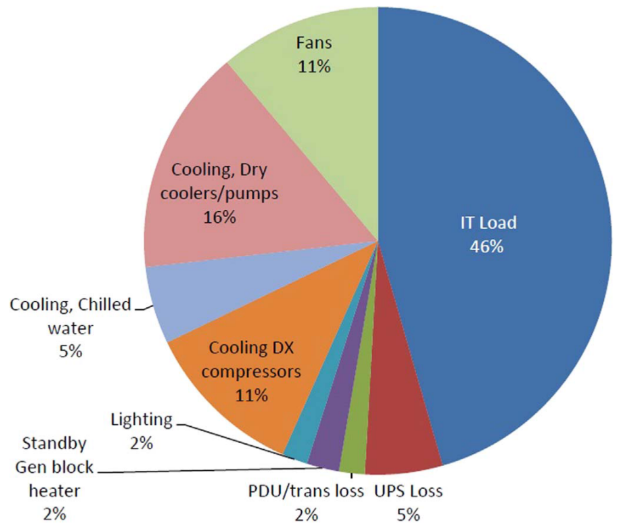


Figure 3: Typical energy distribution of a data center. [18]

$$\text{PUE} = \frac{\text{Total Facility Power}}{\text{IT Equipment Power}}$$

It measures the ratio of overhead power and the actual power needed for computing. A value of one would therefore mean that all energy is used for computing, whereas a data center with a value of two would need twice as much energy for lighting, cooling or safety-related electrical equipment as for computing. In the U.S., the average is about 1.91, and ”green” data centers are generally considered to have a PUE of 1.5 or less, with Facebook achieving as high as 1.1 in one of its modern data centers. [15, p. 2]

However, PUE also has known pitfalls, as the following example shows. If 100 servers were idle in a data center and one wanted to save energy by switching off servers, the PUE would increase with each server switched off, since the energy share of light and cooling would increase. If the number n of active servers were to run towards zero, the PUE would even diverge towards infinity. A second problem is that the PUE does not take into account the efficiency of the deployed servers themselves. One approach to solving these two problems would be to consider power per watt in addition to PUE. The question here is how best to measure performance. One common method is FLOPS (floating point operations per second). The Green500 even lists the 500 most efficient supercomputers based on this metric, although this benchmark is hardly applicable to real production environments. For a fair comparison, the location and climatic conditions must be taken into account in addition to the combination of PUE and performance per watt.

Although energy efficiency plays a major role, it should not be the only factor that determines whether a data center or company is green. In a report published in 2017 [9], Greenpeace examined the world's largest digital companies in terms of their data centers. Decisive factors included energy mix, transparency, efficiency and further commitment. The evaluation shows that, how green a data center is appears to depend heavily on company policy rather than economic reasons, with the majority scoring poorly. Leading companies such as Google, Apple, Facebook, eBay, and Switch are using their influence to get vendors, utilities, and governments to provide access to renewable energy where none existed before. [9, p. 6] Though, this trend seems to take place mainly in the USA by major internet companies. Video streaming should also be highlighted, where all providers except Google's YouTube performed poorly, such as Netflix or Amazon Prime, even though video streaming accounted for two-thirds of global data traffic in 2015. [9, p. 7]

4. CARBON FOOTPRINT OF COMPUTING WORKLOADS

While we addressed the environmental impact of a data center in section 3, we will go just one level further and focus on workloads. If you can quantify the environmental impact of a data center, it can be mapped to a workload, for example, by dividing it evenly among all workloads. Yet, in this paper we focus on the direct environmental costs, more precisely the power consumption. To this end, we first clarify the impact of the utilization of various resources on power consumption, followed by approaches to measure power on a given server and the distribution of idle time across all workloads, and finally present a formula that quantifies the power consumption of a workload on all servers involved.

4.1 Hardware Utilization and corresponding Power Usage

In non-optimized systems, as much as 60% of peak power is wasted in idle states. Furthermore, typically, server utilization is below 30%. [17, p. 1] Consequently, physical machines lose a lot of efficiency due to low CPU load or even idling. It is obvious to consider these idle states as a starting point for optimization strategies (more in Section 5). In some data centers there will be times when the load is regularly minimal, for example at night. In this case one could relatively easy apply optimization strategies like Consolidation (minimize amount of servers) or Power Napping (shut down server in idle state). The problem gets more sophisticated, when idle periods last only a few seconds, although occurring regularly. [17, p. 1]

In the following, the most important hardware components like CPU, disk as well as the system board are examined for their relation between utilization and energy consumption. Figure 4 shows an exemplary power consumption of a server. The whole pie chart represents the maximal energy consumption, and the right, grey side of the pie shows the idle power consumption, which is about half of the peak power. The Idle Power is further divided into its components

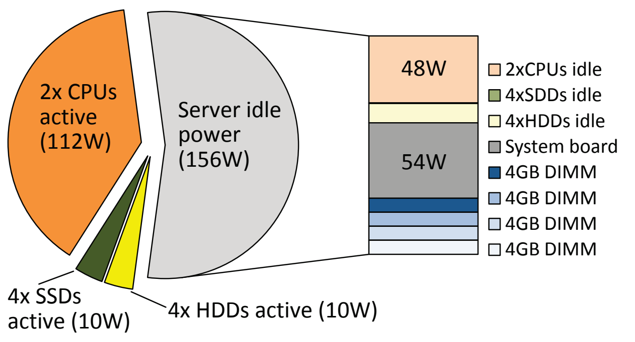


Figure 4: Exemplary power breakdown of one server from [25, p. 2]

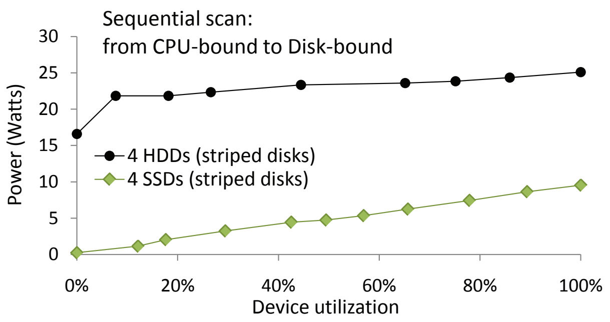


Figure 5: Power consumption in relation to utilization of storage disks. [25, p. 5]

on the right side. The two CPUs waste the most power in idle, followed by the RAM, which needs to refresh its memory in idle, and system board components. The idle power of the HDDs is relatively low whereas the consumption of SSDs is almost zero. On the left half of the pie chart, the additional power consumption when all CPUs and HDDs are fully loaded is illustrated. While the storage media require a similar amount of power, the CPUs consume additional 112 Watt. [25, p. 2] The RAM and the system board do not increase in their power usage with increased system load. While some components have almost fixed power consumption without much difference between load and idle state, SSDs have perfect energy proportionality. As can be seen in figure 5, HDDs do not have this property, as their disk must first accelerate to a certain RPM, which is maintained for some time during inactivity.

In the literature, a linear correlation between CPU load and energy consumption is often assumed. But as shown in figure 6, this is not invariably the case.

According to Dimitris Tsirogiannis et al., three conditions must be met for a linear correlation to be fulfilled. [25, p. 5]

1. operations are cpu-bound
2. there are no shared resources among CPU cores
3. no power management techniques are applied

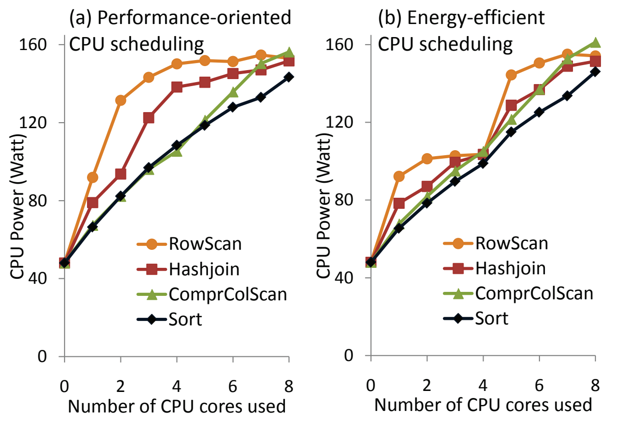


Figure 6: Dual CPU power consumption with database operations. [25, p. 2]

Energy waste concerns at idle have led to calls for a fundamental redesign of computer system components to make energy consumption proportional to workload. Dynamic frequency and voltage scaling (DVFS) of processors is an example of the concept of energy proportionality, allowing up to cubic energy savings at reduced loads. However, processors continue to decrease as a percentage of total server power and DVFS continues to be an active research topic [17].

4.2 Power Consumption of a workload

Mathematically speaking, we understand a workload w as a set of processes that can be executed on arbitrary numbers of computing units(like server, vm, container) simultaneously.

$w := \{p_1, \dots, p_n\}$	: workload of n processes
$pow(p, t)$	: power(watt) of p at time t

Evidently, the simplest case is that only one workload is executed on one or more computing units. In this case, the total power consumption is equal to that of the workload. But when we consider a set of workloads running at the same time on the same resources, the problem becomes more challenging.

$$W := \{w_1, \dots, w_k\} \qquad : \text{all workloads}$$

To calculate the power consumption c of w over a time period $[a, b] \subset \mathbb{R}_+$:

$$c(w, a, b) := \int_a^b \sum_{p \in w} pow(p, t)\, dt$$

However, the power function pow must be continuous, which can be achieved by deriving a function from the hardware. For example by measuring one hardware component under low load, then execute the workload and measure it again. Then, a suitable function can be set up for each hardware component, taking into account the points mentioned in the previous section. These can then be aggregated into a server-dependent function pow. This must be done for each server on its own, when the hardware components differ.

Hardware usually allows not only the load of a process but also the power consumption to be read out. This saves us from creating models that convert utilization to power consumption, which would be not only complicated but also inaccurate. One software that can read the power consumption directly is powerTOP. Assuming that we perform a measurement every second with powerTOP, we can now replace the continuous function pow with the discrete function $powC$, which represents the measured power consumption in second t. This makes the formula c much simpler.

$$c(w, a, b) := \sum_{t=a}^{b} \sum_{p \in w} powC(p, t) \qquad a, b \in \mathbb{N}, a \leq b$$

It makes at least as much sense to consider power consumption as a function of time as it does to consider it as a function of a request. This makes sense especially for e.g. web servers or CDNs. The easiest way to do so is by counting the requests from start time a until b and to divide the power consumption by the amount next. To keep it more general, we will stick with time dependence.

4.3 Base Power

The base power consumption poses a problem when calculating the power consumption of a workload as one has to decide on how to distribute the base power consumption on all current workloads.

Therefore, we adapt function c to also distribute the base power to the workloads. To do this, we first need the total power of the computing units.

In the following, we will neglect the parameters a and b for the sake of clarity. It should be noted that all functions still depend on a time interval.

c_{total}	: total power consumption
$c_{workloads} = \sum_{w \in W} c(w)$	: consumption of all workloads

Splitting it evenly between all k workloads is simple:

$$c_{split}(w) := c(w) + \frac{c_{total} - c_{workloads}}{k}$$

But since the power and computation time needed is not equal among all workloads, c_{new} is not very fair. It would even assign energy to completely inactive workloads. Another approach would be to distribute it based on workload run time. But as we already have the measured power consumption of a workload, we think it is the fairest one to use:

$$c_{fairsplit}(w) := c(w) + \frac{(c_{total} - c_{workloads})c(w)}{c_{workloads}}$$

$c_{fairsplit}$ splits the energy overhead proportionally to the energy consumption of a single workload and is thus much more accurate than c_{split}.

4.4 Calculating the Carbon Footprint

With base power considered, we can now calculate the carbon footprint CF of one workload by applying the carbon intensity I (in grams / kilowatt) of the data center:

$$CF = c_{fairsplit} * I$$

Note that we only included the energy, because the energy is easy to assign to one workload and usually the primary source of carbon footprint for one workload. Future work could start here and calculate the CO2 footprint of consumed resources of a data center and distribute it to the workloads. However, since the resulting footprints are likely to differ significantly among data centers, this task probably lies with the operators of the centers themselves.

5. OPTIMIZATION STRATEGIES

Given how complex the structure of a data center can be, nearly every component of it can be optimized to reduce its overall power consumption and in turn, the power needed to execute a given workload and the associated carbon footprint. This section will focus on some of the efforts made to save power, from large-scale architectural redesigns up to workload-specific strategies.

Not every strategy is applicable to every business, for example, there will usually be some sort of trade-off between energy consumption and latency, which often has a direct impact on revenue. Also providers who rent out servers cannot necessarily optimize well, as they have to guarantee a certain performance, especially dedicated servers elude most optimization strategies here. So much for possible pitfalls, which we will not go into further here, as in this section we refer in particular to companies that operate or rent highly scalable data centers.

5.1 Choice of Location

As previously discussed, cooling consumes more than 40% of a data center's power, so reducing the need for it is a reliable way to save power. Besides more advanced cooling systems or a higher server temperature, one can also take advantage of environmental cooling by placing the clusters at high altitudes, near water reservoirs or other naturally cold places [21]. While this does reduce cooling requirements, it also leads to logistic problems, as those places tend to be remote. Proper care should be applied to choose a suitable, easy to reach location for each data center and to ensure that it makes the most out of the features the environment provides. How these are utilized is discussed in more detail in the next section.

5.2 The right Temperature

The temperature to which a data center should be cooled is a matter of controversy. On the one hand, it is claimed that a room temperature of 28°C instead of 20°C does not harm servers and reduces the power requirement of the cooling system by more than one fifth [26]. On the other hand, operators fear indirect costs due to higher failure rates and shorter component lifetimes. Simply increasing the ambient temperature in the data center may not always have the desired effect of reducing energy consumption. [19, p. 1] The optimum temperature varies from data center to data center and must always be found out anew, nevertheless there are guideline values that an efficient data center should fulfill. The trend is toward warmer temperatures. While older data centers were generally designed for a temperature of 15 or 16°C, loosely based on the principle that colder is better [19, p. 2], many data centers today run a few degrees warmer than they did 10 or 15 years ago to save on cooling costs [23], and while ASHRAE (American Society of Heating, Refrigerating and Air-Conditioning Engineers) recommended a temperature of 20 - 25°C in its first edition of Thermal Guidelines for Data Processing Environments in 2004, by 2008 it was 18 - 27°C. Meanwhile, there are different classes depending on factors such as humidity and more, allowing a temperature range of up to 5 - 45°C. [22, pp. 10-11], which means that filtered outside air or water can be used for cooling without air conditioning refrigeration or heat exchangers. The generated heat is oftentimes lost and simply emitted to the outside. Yet data centers have a lot of savings potential here. Especially in cold regions, where data centers like to be built anyway, this circumstance can be used well as district heating. Sweden is considered a pioneer in this area. A data center can theoretically heat up to 10,000 homes [16]. If the infrastructure for this is lacking, however, energy can also be recovered from the heat.

5.3 Power-Efficient Hardware

Increasing the efficiency of the hardware used for computing will also lead to a reduction in overall power consumption. Numerous CPU manufacturers have already implemented saving strategies into their products: Intel introduced the "SpeedStep" technology, which dynamically allocates power based on the current workload. AMD invented PowerNow to reduce CPU Voltage and clock speed if the workload is low where as Cool n' Quiet processor's only reduce them while the CPU is idle. [21] On the matter of storage devices, the usage of Solid State storage instead of hard disk storage is also advised, as they requires less power for the sole reason of not having moving parts. Hardware manufacturers such as IBM are also working on improving their storage devices to reduce thermal waste and increase efficiency at higher temperatures. [7, pp.3-4]. The usage of low power/single board computers can also lead to an increase in efficiency. [12]

5.4 PowerNap

Given that idle states are major "power wasters" (as discussed in section 4.1), reducing the energy consumption during these phases is indispensable for power saving. As such, multiple approaches regarding it have already been proposed, with PowerNap being one of those. It's base premise is rather simple - work at full power while occupied, rest while idle and return to full power as soon as new work is detected, as the system is completely inactionable during the resting phase in order to conserve as much power as possible. Thus, the average powerP_{avg} consumed in a system can be described as

$$P_{avg} = P_{nap} * F_{nap} + P_{max}(1 - F_{nap})$$

where P_{nap} and P_{max} describe the power used in inactive and active mode respectively and F_{nap} the time spent in the resting phase. The main factor that distinguishes PowerNap from similar implementations is the addition of additional "wake" and "suspend" states. These act as additional jobs, performed when new work batches arrive and the system is in its resting state and when the job queue is empty for "wake" and "suspend" respectively. These transitions have to be performed extremely quickly (10 ms or lower), otherwise much of the potential energy savings are lost. At peak performance, with transition speeds lower than 1 ms, power demand rises almost linearly with usage, eliminating

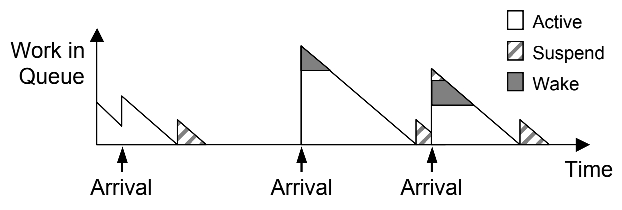


Figure 7: Example sequence of events occurring in a PowerNap-managed system [17]

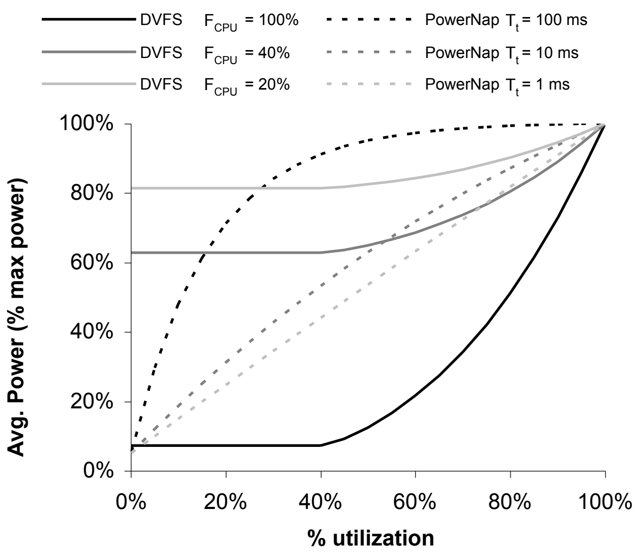


Figure 8: PowerNap and DVFS(another power-saving method) compared by their power demand based on usage [17]

the issue of idling completely. But even sub-optimal implementations such as ones with a transition speed of less than 10 ms still show substantial power savings. Implementing PowerNap could reduce power usage by almost 70%, which can be further improved by the usage of advanced power delivery systems. [17]

5.5 Cluster Management

The ever expanding IT sector and the increasing demand in cloud computing services has caused a number of different cluster types to emerge. One of these are HDFS-Clusters (Hadoop Distributed File System). In a HDFS cluster, data and workloads are balanced between all servers so that any server can be used for any job at any given time. This, however, makes it hard to nigh impossible to take advantage of switching into low-power modes while idle by methods such as the aforementioned PowerNap as switching between active and inactive states takes time and thus require long, predictable idle phases which generally do not occur in HDFS-managed clusters. GreenHDFS promises to be an energy-efficient replacement for the existing HDFS and offers a solution to the problem described above by dividing the cluster beforehand into low-demand (cold) and high-demand (hot) zones.

Hot zones contain relevant, frequently used data and therefore are optimized towards performance and speed. Power is not a concern in this zone and thus no efforts regarding power saving are being made.

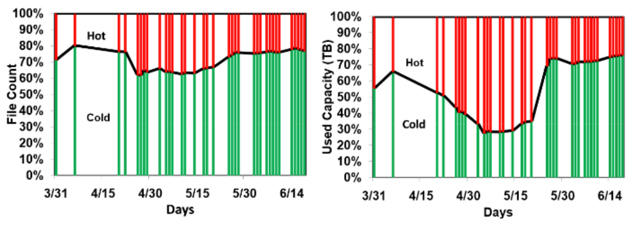


Figure 9: Percentage of hot/cold data and the associated file count of an Yahoo! cluster [13]

Cold zones on the other hand contain old data which is rarely used. It makes up more than 60% of the total data, therefore the servers responsible for holding it consist mostly out of storage media with little or no regards payed towards performance. As such, the physical space needed for the cold zone to handle this data is rather low, which in turn improves the performance of the hot zone due to the ability to assign more servers to it. This allows for significant power saving by assigning old data to the cold zone and shifting these into constant low-power mode, waking them up only if the stored data is needed or to use the latent computing capacity to temporarily boost performance during periods of high utilization.[13] The usage of scheduling tools like Oozie can further reduce power consumption, since they can preemptively start idle servers when a workload is scheduled there, run all jobs which require the data stored there at once and power them down again thereafter, preventing constant switching and performance loss due to wake-up latency.

A test using GreenHDFS was performed on an Yahoo!-cluster and showed direct power savings of 26 percent (41607$ worth of electricity) in comparison to the standard HDFS, not accounting the indirect power savings due to a reduced temperature in the cold zone and the associated reduction in cooling demand.[13]

5.6 Workload Shifting

In big networks, consisting of a number of geographically distributed data centers, Workload Shifting can be used to basically execute a given workload where it would produce the least amount of carbon emissions. However, simply assigning every request to one data center would not only lead to a spike in latency due to increased distance but also flood it with requests, creating a situation similar to a DDoS attack. Therefore, the average response time of a data center is used in conjunction with its carbon density to determine the optimal place for each workload.

In order to test the applications of Workload Shifting, a partial simulation of the *Amazon* network was conducted [27], spanning three data centers located in California, Virginia and Dublin respectively, with request sources in the vicinity of the data center. The amount of requests originating from each of those was calculated using *Facebook* data, as their services cover a broad spectrum and are used globally. Carbon intensity was calculated using the energy mix of a data center's location while considering the availability of solar power, the renewable energy of choice due to its predictable power output and downtime pattern.

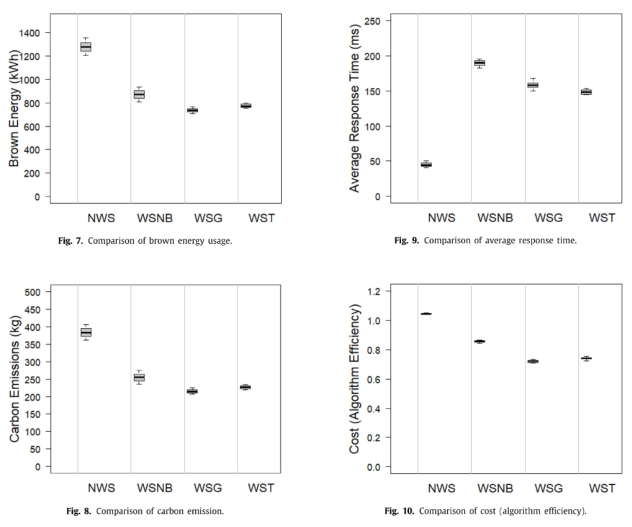


Figure 10: Results of the simulation. WSG is the main algorithm, WSNB and WST are variations of it and NWS doesn't utilize Workload Shifting[27]

The results of the simulation show that in comparison to NWS, brown energy usage and therefore carbon emissions have dropped while the average response time has increased significantly, with small differences between the variations of the Workload Shifting algorithm. Taking this into account, Workload Shifting proves to be an effective way to reduce carbon emissions, if latency isn't a concern [27]. Because it increases latency, it may not be fit for every kind of workload: Urgent jobs which can not afford to wait shouldn't be handled by a workload shifting algorithm to ensure their execution in a timely manner.

6. CONCLUSIONS

There are four factors - power consumption, carbon intensity of the power used, owner power policy and the environmental cost of the used components - influencing the carbon footprint of a data center, the main one being power consumption. Using it to calculate the proportion of a given workload yet proves to be a challenge as not all of the total power is used for computing. In fact, over half of it is consumed by cooling and lighting operations, which shouldn't be included in calculation since they occur independent of the current workload. Also, power is still used in idle states, further complicating the process. One has to consider this to get an accurate result. These can then be obtained by employing tools like PowerTOP and the formula c mentioned in section 4.2. If several workloads are to be considered, base power has to be divided among them. The results can then be translated to kWh and multiplied with carbon intensity to get the carbon footprint. On the matter of reducing carbon emissions, there is no "silver bullet" to universally solve the problem. A data center's carbon footprint can be reduced and optimized in a multitude of different ways, some of which differ based on factors like location or management model used. One could, however, compare these and create a list of priorities based on their overall significance: Some approaches promise huge amounts of saved power (e.g. Cooling/Idling optimizations) or are easy to implement (e.g. using more advanced components for server construction), while other methods may lack efficiency, are harder/more expensive to implement or come with major trade-offs (e.g. Increased response time in Workload Shifting) compared to others.

7. ACKNOWLEDGMENTS

This paper was made using Overleaf.

8. REFERENCES

[1] Bilanz 2019: Co2-emissionen pro kilowattstunde strom sinken weiter | umweltbundesamt. `https://www.umweltbundesamt.de/presse/pressemitteilungen/bilanz-2019-co2-emissionen-pro-kilowattstunde-strom`. Accessed on 2021-05-30.

[2] Data & statistics - iea. `https://www.iea.org/data-and-statistics/data-browser/?country=WORLD&fuel=Energy%20consumption&indicator=TotElecCons`. Accessed on 2021-05-16.

[3] Entwicklung der spezifischen kohlendioxid-emissionen des deutschen strommixes | umweltbundesamt. `https://www.umweltbundesamt.de/bild/entwicklung-der-spezifischen-kohlendioxid`. Accessed on 2021-05-30.

[4] Frequently asked questions (faqs) - u.s. energy information administration (eia). `https://www.eia.gov/tools/faqs/faq.php?id=74&t=11`.

[5] Jonathan koomey: Blog. `https://www.koomey.com/post/153838038643`. Accessed on 2021-06-01.

[6] A. S. G. Andrae and T. Edler. On global electricity usage of communication technology: Trends to 2030. *Challenges*, 6(1):117–157, 2015.

[7] A. D. Borah, D. Muchahary, S. K. Singh, and J. Borah. Power saving strategies in green cloud computing systems. *International Journal of Grid Distribution Computing*, 8(1):299–306, 2015.

[8] U. Cisco. Cisco annual internet report (2018–2023) white paper, 2020.

[9] G. Cook, J. Lee, T. Tsai, A. Kong, J. Deans, B. Johnson, and E. Jardim. Clicking clean: Who is winning the race to build a green internet? *Greenpeace Inc., Washington, DC*, 5, 2017.

[10] D. R. Danilak. Why energy is a big and rapidly growing problem for data centers. Accessed on 2021-05-28.

[11] A. Feldmann, O. Gasser, F. Lichtblau, E. Pujol, I. Poese, C. Dietzel, D. Wagner, M. Wichtlhuber, J. Tapiador, N. Vallina-Rodriguez, O. Hohlfeld, and G. Smaragdakis. Implications of the covid-19 pandemic on the internet traffic. In *Broadband Coverage in Germany; 15th ITG-Symposium*, pages 1–5, 2021.

[12] M. D. Görtz, R. Kühn, O. Zietek, R. Bernhard, M. Bulinski, D. Duman, B. Freisen, U. Jentsch, T. Klöppner, D. Popovic, and L. Xu. Energy efficiency of a low power hardware cluster for high performance computing. In M. Eibl and M. Gaedke, editors, *INFORMATIK 2017*, pages 2537–2548. Gesellschaft für Informatik, Bonn, 2017.

[13] R. T. Kaushik and M. Bhandarkar. Greenhdfs: towards an energy-conserving, storage-efficient, hybrid hadoop compute cluster. In *Proceedings of the USENIX annual technical conference*, volume 109, page 34, 2010.

[14] J. Koomey, S. Berard, M. Sanchez, and H. Wong. Implications of historical trends in the electrical efficiency of computing. *IEEE Annals of the History of Computing*, 33(3):46–54, 2011.

[15] J. Li, J. Jurasz, H. Li, W.-Q. Tao, Y. Duan, and J. Yan. A new indicator for a fair comparison on the energy performance of data centers. *Applied Energy*, 276:115497, 2020.

[16] mdr.de. Ein rechenzentrum könnte 10.000 wohnungen heizen | mdr.de. `https://www.mdr.de/wissen/faszination-technik/stromverbrauch-rechenzentren-koennen-heizen-und-kuehlen100.html`. Accessed on 2021-05-29.

[17] D. Meisner, B. T. Gold, and T. F. Wenisch. Powernap: eliminating server idle power. *ACM SIGARCH Computer Architecture News*, 37(1):205–216, 2009.

[18] J. Ni and X. Bai. A review of air conditioning energy performance in data centers. *Renewable and Sustainable Energy Reviews*, 67:625–640, 2017.

[19] M. K. Patterson. The effect of data center temperature on energy efficiency. In *2008 11th Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems*, pages 1167–1174, 2008.

[20] R. Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, 1997.

[21] S. Singh, A. Swaroop, A. Kumar, and Anamika. A survey on techniques to achive energy efficiency in cloud computing. In *2016 International Conference on Computing, Communication and Automation (ICCCA)*, pages 1281–1285, 2016.

[22] A. TC et al. Data center power equipment thermal guidelines and best practices. *ASHRAE TC 9.9, ASHRAE, USA*, 2016.

[23] P. Thibodeau. It's getting warmer in some data centers | computerworld. `https://www.computerworld.com/article/2483971/it-s-getting-warmer-in-some-data-centers.html`. Accessed on 2021-05-29.

[24] P. Thurrott. Amd delivers a major mobile efficiency milestone - thurrott.com. `https://www.thurrott.com/hardware/236987/amd-delivers-a-major-mobile-efficiency-milestone`, 2020. Accessed on 2021-06-01.

[25] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Analyzing the energy efficiency of a database server. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 231–242, 2010.

[26] U. (UBA) and B. Engel. Vortrag gut für ihr rechenzentrum. gut für die umwelt.. ”`https://www.blauer-engel.de/sites/default/files/publication/2017-10-10-prasentation-der-blaue-engel-fuer-rechenzentren-final-2-folien-pro-seite.pdf`”, 2014. Accessed on 2021-05-25.

[27] M. Xu and R. Buyya. Managing renewable energy and carbon footprint in multi-cloud computing environments. *Journal of Parallel and Distributed Computing*, 135:191–202, 2020.

Project management of data-driven projects - Challenges and Management Approaches

Nedelcho Dimov
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
nedelcho.dimov@rwth-aachen.de

Ivan Slavov
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
ivan.slavov@rwth-aachen.de

ABSTRACT

More and more organizations have an interest to invest in data-driven projects in order to keep pace with their competition. Because of this trend, many studies and experiments started to focus on the exploration of this field and its improvement for example seeking more efficient algorithms and data storage. Many experts say that there is a lack of research on how the teams should be managed, coordinated and formed. The high failure and incompletion rate of such projects confirm that an in-depth research on project management of data-driven projects is needed.

In this paper we conduct a literature review where we look for the main challenges that the team members, managers of the projects and other participants in such projects meet. We discuss them and show the causes for them. The literature review is conducted to find approaches for the overcoming of the challenges mentioned in this paper.

The results of the literature review are the following problems in data-driven projects: need for a better definition of lifecycle development process, process methodology and concrete team roles. The approaches we found in the literature bring with them some advantages but also some disadvantages that we discuss in the paper. Other approaches we found need to be tested in real projects in order to be evaluated so that their positive and negative effects can be explored.

Keywords

Process methodologies, management challenges, management approaches, big data, data science, data mining

1. INTRODUCTION

Data-driven projects are projects in the fields of big data, data science, data mining. There are such projects in big companies like Facebook, Twitter, LinkedIn [1], where big amount of data should be processed.
Big data is related to data science, but in big data the quantity of data collected and analysed is far greater. Therefore a project is called a big data project, when large amount of data should be processed and analysed. Furthermore, different tools, computing infrastructure and storage are needed to process and analyse such a huge amount of data. For a project to be considered as a big data project it should fulfill the 4 V's namely: **Variety, Volume, Velocity and Veracity**. The term variety describes that there are many different types of data. Volume means that we work with a large amount of data. Velocity describes the speed with which we process the data. Veracity means that the data is trustful and of high quality [1] [2].
Data Mining projects are another example for data-driven projects, where a huge amount of data is analysed. The aim of the team in such projects is to find interesting and valuable structures within the data. These structures can be global or local as the analysis of which one will be researched depends on the interest in the specific field. It uses tools from many fields for example from computer science, machine learning etc. [19]
The last type of data-driven projects that we mention in our paper are the Data Science projects which consist of many principles helping the teams to extract information and knowledge from data. In such projects the teams use Data Mining but apart from it there are other things that they use. For example causal analysis, methods and methodology for visualizing data etc. [20]
There has been a high interest in data-driven projects, but according to some studies a large amount of such projects fail [1]. One of the main reasons is that there is no unified standard for the project management of data-driven projects [2]. When we look at the research in the field of big data, data science and data mining, we can clearly see that more of them are exploring algorithms and data models and give suggestions about their improvement. Very little research is about the project management of data-driven projects and gives suggestions about overcoming the problems in this field [3]. Therefore in this paper we will perform a literature review to identify the challenges for the project management in this field and look for some suggestions for overcoming mentioned challenges.

The aim of this paper is to find possible solutions for the following research questions:
1. What are the main challenges in data-driven projects?
2. What approaches exist for overcoming said challenges?
3. What are unsolved challenges?
In the following, the discovered challenges will be mentioned. After that, possible approaches for them will be presented. At the end there will be a summary of the challenges that

have not been solved yet.

2. METHODOLOGY

The literature review was conducted in the following steps:

First, we searched in IEEE explore for papers with similar keywords like process methodologies challenges, data-driven projects, big data, data science, data mining challenges. We restricted our search to challenges related to the project management of such projects.

After that, we searched for approaches in Google Scholar and IEEE explore with similar keywords as above. We focused on the papers that suggested approaches for challenges that we mentioned.

We used the results and findings of 26 papers or articles about the challenges and approaches in such projects.

Finally, we summarized all the information.

3. PROJECT MANAGEMENT CHALLENGES

In this section we will talk about the challenges in data-driven projects. We found the following challenges during the literature review :

3.1 No concrete definition of roles in data-driven projects
3.2 Difficulty of choosing the right process methodology
  3.2.1 Unclear requirements
  3.2.2 Teamwork and communication problems
  3.2.3 Plannability problems
3.3 Fulfilling the 4 V's

3.1 No concrete definition of roles in data-driven project teams

The first big challenge with managing such projects we found, was that there is no concrete definition of roles within a team. This is because in data-driven projects, the jobs that are required are often from different fields and require different skills. For example, many of the projects need people that are specialized in Big Data, Data Mining and Data Science [22]. Team members in such projects need many different skills in the disciplines computing, math, statistics, machine learning and business domain knowledge[24].

Role	Number of Jobs
Data Science Researcher	0
Data Scientist	440
Data Architect	467
Data Analyst	696
Data Science Programmer	0
Data Engineer	378
Big Data Engineer	107
Information Architect	123
Data Science Manager	7
System Orchestrator	0
Data Provider	28
Big Data Application Provider	0
Big Data Framework Provider	0
Data Consumer	3
Security and Privacy	148

Figure 1. Search results for a respective job [4]

In our paper, we will reference the work of Jeffrey Saltz and Nancy Grady [4] who researched the roles defined by 5 different organizations - NIST, EDISON, SAIC, Springboard and Gartner. In these organizations, they found that there are positions with similar names that have different tasks or there are different names for positions that have the same task.

The authors of this paper conducted an experiment by searching on a jobs website for some positions, used in the above-mentioned organizations. The results of the experiment can be seen in figure 1.

As we can see some of the roles defined by the organizations are not even used in the real world and other are used too broadly. This makes it very difficult to define the skills needed for each of those roles and employers, who are looking for employees in the field, cannot make a right judgement, if the candidates are qualified enough for the respective job. In the approach section we will briefly describe the roles defined by the five organizations. Then we will compare these roles and talk about some of the skills that they require.

3.2 Difficulty of choosing a process methodology

Another challenge we found in the literature is that there are many different methodologies and each project needs a specific process methodology depending on the fields that are used in it. Furthermore, the methodologies used in practice are defined only for a specific field or have not been adapted for the new challenges that data-driven projects bring [10] [9]. Therefore, their application in real world projects decreases [10]. Companies search for new methodologies that are more appropriate or use ones that are defined for another field but are more efficient than the current methodologies. Some examples are Agile Kanban, Scrum , CRISP-DM [9].

We also found that every known methodology brings not only advantages but also disadvantages, which makes choosing the right process methodology tremendously important for a successful project. One reason is that in such projects the amount of data used is too large and many different fields are involved. One person cannot handle them, which leads to the creation of teams to process the data [22] [11]. This challenge is the cause for further challenges project managers of data-driven projects face. The first one is that requirements given by the customers are not clearly defined. The second one is that there are problems with teamwork and communication in such teams. The last one is that it is hard for the teams to plan the future work. Now we will show about each of those problems in more detail.

3.2.1 Unclear requirements

According to the paper by Jeffrey Saltz and Ivan Shamshurin [2] there are more unclear requirements in data-driven projects compared to other software development projects. One of the main reasons for the uncertainty is that the analysis in data-driven projects has exploratory nature. Because of this nature, the requirements in such projects are not clear and the results from the projects about these requirements cannot be validated very often. The focus of team processing the data and business teams are different. For example, business teams focuses on financial benefits, while teams, processing the data, tend to focus on model accuracy [23]. One more reason for that is that there is a gap between the customer and project manager caused by unwillingness to collaborate. It is a big problem, because software development teams do not have insight. That is why, they need domain experts to help them [23]. This makes it harder for the team to understand and clarify the requirements of the project and then successfully fulfill them. Apart from that, customers do not understand how data-driven projects work and have too high expectations [22]. The wrong understanding of the requirements, that is a specific characteristic for the data-driven projects is one of the main reasons for the need of a specific methodology for such projects to be created [3].

3.2.2 Teamwork and communication problems

Another problem resulting from the lack of appropriate process methodology is that around three quarters of data-driven projects fail because of bad communication in the team [3]. A cause for that can be that the teams of such projects should consist of team members with different skills and from different areas. For example, business teams and technical teams in a project can have different metrics according to which they assess the success of the project. It can happen that while one of the teams assesses the project as successful the other one can assess it as unsuccessful [23]. Another example can be the collaboration between the software development team and data science team in the project. They have to understand how the project will be used in production in order for the software development team to integrate it correctly and the data science to understand the real requirements about the project[23].
Therefore, effective communication in such teams is very important factor for the success of the project. If such problems arise in the projects they can lead to delays. It is advisable for a methodology to be created, where this aspect is considered and guidelines are given on how to solve such problems within the team [21].

3.2.3 Plannability problems

The explorative nature of analysis also causes the last challenge that we will mention in our paper. Data-driven projects need to respond on changes in the requirements at any time, because they are very common especially during the analysis process. These changes can be related to some problems that can arise in this process. Such changes can happen because the model or the data change. In such cases an urgent reaction is needed so that the changes can be made in time and so that the analysis process can be accomplished in time [2].

In the approach section, we would like to summarize what methodologies exist and can be applied to data-driven projects so that these challenges can be overcome. Current data-driven projects use some of the following methodologies: CRISP-DM, Agile Kanban and Scrum, but they have some drawbacks. Some proposed approaches we found are for example Team Data Science Process (TDSP), Knowledge Discovery in Data Science (KDDS) and Refined Scrum-DS. They can be possible solutions to the challenges brought by the other methodologies in a data-driven project. Furthermore, we will describe these methodologies and we will discuss some of their strengths and weaknesses. Apart from that, we will compare the different methodologies and try to relate them to the problems introduced above.

3.3 Fulfilling the 4 V's

Another challenge we found in some papers is fulfilling the 4 V's of big data. As mentioned in the introduction there is a high failure rate of such projects. It is 55%, where general software projects only have about 38% failure rate [18]. The most essential requirement in big data projects is to fulfill the 4V's, but there is no standardized process that considers them and is one of the main causes for their great percentage of incompletion. Variety is the most difficult out of the 4 V's that the manager of the project needs to deal with[18]. The paper The Design of a Software Engineering Lifecycle Process for Big Data Projects by Yen-Tai Lin and Sun-Jen Huang [18] proposes a standard that can handle the challenges we mentioned above. In the approach section section we will describe it briefly.

4. PROJECT MANAGEMENT APPROACHES

In this section we will talk about some approaches we found, that try to solve some of the above-mentioned challenges.

4.1 Approaches for the definition of roles in data-driven projects

4.1.1 NIST

The NIST organization is a standards organization. They have defined the following roles: **System Orchestrator, Data Provider, Big Data Application Provider, Big Data Framework Provider, Data Consumer, Security and Privacy and Management** .

System Orchestrator: defines the goals of the project and monitors the team.
Data Provider: gives new data or information feeds.
Big Data Application Provider: gives instructions how the project should be executed to meet the requirements set by the System Orchestrator and these related to the security and privacy.
Big Data Framework Provider: creates the framework where the data will be processed to meet all privacy and security requirements.
Data Consumer: this is the end user or system.
Security and Privacy: monitors the privacy and security requirements.
Management: manages the system and data-driven lifecycle process[4].

4.1.2 EDISON

The EDISON organization is another standards organization that has defined the following roles: **Data Scientist, Data Science Researcher, Data Science Architect, Data Science Programmer, Data/Business Analyst**.

Data Scientist: finds data sources and applies mathematical models to process the data.
Data Science Researcher: finds valuable knowledge and reveals relations between different goals of the project.
Data Science Architect: creates and maintains the architecture of data-driven applications.
Data Science Programmer: codes an analytics application.
Data/Business Analyst: analyses the found data and makes

it usable for presenting[4].

4.1.3 SAIC

The SAIC organization is an industry organization that has defined the following roles: **Information Architect, Data Scientist, Metrics and Data, Knowledge and Collaboration Engineer and Big Data Engineer**.

Information Architect: develops data models, data standards and the design of data structures.
Data Scientist: works in all stages of the life-cycle process, extracts value from the data and cares for the verification of the received results in each step.
Metrics and Data: creates data models to increase the productivity.
Knowledge and Collaboration Engineer: promotes knowledge management and collaboration within the enterprise.
Big Data Engineer: role that covers a big part of the tasks within the life-cycle of the project[4].

4.1.4 Springboard

The Springboard organization is also an industry organization. They have defined the roles: **Data Engineer, Data Scientist, Data Analyst and Data Architect**.

Data Engineer: knows many programming languages and implements requests from data scientists.
Data Scientist: transforms the goal of the project into a data question and creates predictive models to find a solution and present it.
Data Analyst: finds the value from the data and shows it in presentable form.
Data Architect: structures the technology needed to manage the data models[4].

4.1.5 Gartner

The Gartner organization is an advisory company that has defined the following roles: **Data Scientist, Data Engineer, Business Experts, Source System Experts, Software Engineers, Quant Geeks and Unicorns**.

Data Scientist: participates in all stages of life-cycle process.
Data Engineer: transforms the data in such a way that data scientists can easily understand and process it and to increase the productivity.
Business Experts: have a good understanding of the business domain.
Source System Experts: know the data at the business application level.
Software Engineers: codes, integrates data or deploys results.
Quant Geeks: very skillful in a specific field and sometimes are needed occasionally and sometimes mandatory.
Unicorns: have a huge variety of skills and with very deep understanding of all data science related topics[4].

After the introduction of the roles in different organizations, we will make a comparison of them.
As we can see above almost all of the organizations have the data scientist role and its definitions are very similar. This job requires skills in the fields of software engineering, math, statistics and data communication. The tasks of this job are management of the data, finding and management of data sources, building mathematical models, translating business goals into data questions etc. This job is among the most used in other organizations and this can be confirmed by figure 1. It is among the most searched jobs in this field.

Another popular position among different organizations is Data Engineer. The main required skills are software engineering, math and statistics. In some organizations the main difference to data scientist is that this job does not require skills related to data communication. Its focus lies on skills related mostly to software engineering like programming in different languages, because they implement the model received by the data engineer. Furthermore, data scientists use machine learning and data analytics, so that skills related to these fields are also required. For data engineers such skills are not necessary, because they deal with data pipeline, data sets and data ingestion[4]. In some of above-mentioned organizations the jobs big data framework provider, data science programmer have similar roles to data engineers, but figure 1 shows that these names are not used at all.

Another position defined in some organizations is data architect. Its tasks are mainly related to the creation and maintenance of facilities and applications used for the management of the data models. This is also popular according to figure 1.

Data analyst is the most common job as we can see in figure 1. It is related to the analysis, visualisation and explanation of the data and requires skills related to data communication, math, statistics and algorithms. In comparison to data scientists software engineering skills are not needed.

The role data provider defined by NIST coincides with a task of the data scientist in EDISON which is provision of data. This job is not popular among the different organizations according to figure 1.

Quant geeks and unicorns are unique roles that are only defined by Gartner. For quant geeks excellent math and statistics skills are required, while unicorns should be good at every required skill needed in such projects.

Data consumer, system orchestrator, management, metrics and data, knowledge and collaboration engineer are roles that are not popular among different organizations and can be seen only in one specific organization.

In the literature, we found no concrete suggestion for the definition of the roles within a data-driven project team. The reason for that is the continuous evolution of these projects [4]. There is currently a need for an unification of the definition of said roles. The roles defined by these organizations can serve as a foundation for the future work on unification of the roles in such projects.

4.2 Approaches for a process methodology

In this section we will discuss the approaches when it comes to choosing a process methodology. First, description of the methodologies will be given. After that, the advantages and disadvantages each of the methodologies will be presented. Lastly, there will be a discussion about which challenges each of the methodologies solve.

4.2.1 CRISP-DM

CRISP-DM (also known as Cross Industry Standard Process for Data Mining) was defined in the 1990s and it consists of 6 steps which form a cycle through which the project can iterat. The steps are: **Business understanding, data understanding, data preparation, modeling, evaluation, deployment**.[9] [12]

Business understanding: this is the first step of the cycle. The main purpose of the stage is to identify the project goals and requirements and they get turned into a problem in the field of data mining.
Data understanding: In this step the team begins to collect data and tries to understand the data and whether problems can arise. They attempt to extract hidden information.
Data preparation: Deals with processing the data and turning it into the final data set
Modeling: In this phase the team picks up a modeling technique and changes the parameters to reach optimal values
Evaluation: The team evaluates the steps of the previous phase and whether the models meet the project goals and requirements.
Deployment: after the creation of the model the team needs to reorganize the results to make them understandable and useful for the end user. [12]

Now that the CRISP-DM methodology has been described, we will discuss a little bit about its real world application. According to reports, the usage of this methodology is decreasing but some experts claim that it would be very useful for the definition of a standard methodology used in all data-driven projects [11]. One major drawback of this methodology is that it is not updated anymore, redefined and transformed to be used in data-driven projects. There is very small number of tools that contain it [10]. This methodology cannot handle the challenges related to the exploratory activities that are a main characteristic of the data-driven projects [25]. It is not flexible enough for such projects and cannot overcome the plannability problems in such projects [25]. This methodology spends much time on understanding the requirements of the project [8]. Thanks to the business understanding step, teams understand the requirement and business goals [12]. This can be confirmed by the experiment of Ivan Shamshurin, Jeffrey Saltz and Kevin Crowston, who conducted an experiment in which they tested different methodologies within teams of students. The students, who used CRISP-DM, had good understanding of the requirements. The team members of the experiment said that this methodology improved the communication, collaboration and teamwork [8].

4.2.2 Agile Kanban

In recent years, a Lean approach has become popular in the field of the software development. It focuses on the effectiveness of the work and how to reduce the unnecessary activities to speed up the process. At first, it has become very popular in the car manufacturing industry and has shown to produce good results. Its principles are: focus on quality, fast results, respect the people, creation of knowledge, always trying to improve the process, reduction of the unnecessary things and activities in the process. Kanban is one of the tools used in the Lean approach. It is used in the project management and is an addition to the agile approach. The most important thing from it is the ”Kanban board”. It represents a list of all the tasks that should be done and who is working on them. In this list, the priority of the tasks can be seen. This helps the team members to see the task which they should do easily and shows them how to deal with it fast. This clarity is achieved because of the following steps:

Preparation: where the team tries to understand the business goals of the project and the data.
Analysis: where the team starts working on the project and during the implementation each task that it has done and has to do is on the board.Also the executed tests are shown on the board.
Deployment: where the team shares feedback and discuss the results.
There is a limitation to the tasks that can be done in each phase. This forces the team to prioritize tasks and to work together. In this way, results can be delivered faster and tends to have better quality. [8] [13]

While we were reviewing the literature, we found a summary of the benefits of the Kanban approach in projects related to software development by Ahmad, M. O., Markkula, J., & Oivo, M. [13]. The experiment done by Ivan Shamshurin, Jeffrey Saltz and Kevin Crowston confirms that these benefits apply to data science which is a part from the data-driven projects [8].

The design of this methodology leads to better overview of how the process should be executed. The challenge about unclear requirements can be solved by Agile Kanban. Because of the ”Kanban board” teams participating in the experiment had a good understanding of the requirements and goals of the project, which according to results of the experiment leads to a higher customer satisfaction. In this methodology, the plannability challenges caused by new requirements of the customer can be handled easily and earlier. This happens, because the members working on the project make decisions what should be done. Furthermore, they decide what are the priorities in the project without the approval of top-level management [13]. This is due to the Kanban board that helps for the prioritization of the tasks [8]. Apart from that, there is no fixed timetable and at any time the new requirements can be considered. Eventually, a change of the priorities can be made [13]. Because of these reasons, Agile Kanban is very flexible and can help for solving the plannability challenges. While we were reviewing the literature, we found the work of Ahmad, M. O., Markkula, J., & Oivo, M. [13], who show some general advantages of this methodology. In real projects, it brings in comparison to another agile methodology faster delivery of the results which are of good quality. It also leads to an increase in learning within the team and to better coordination and communication with other people involved in the project. This methodology increases the self-organization and motivation of the team members. This happens, because they can see what other members prefer to work on and approve or reject their preference. Complex tasks and individual struggles of a team member to deal with a specific task can be noticed faster, easier so that team members can get help earlier [13].

4.2.3 Scrum

The Scrum methodology is one of the most used agile methodologies in software development [5]. It enables the team to be very adaptive to changing requirements of the customers and brings results faster. It represents a cycle through which a team iterates and conducts tests continuously. It is designed to speed up the process and enhances the communication in all phases of the project. The Scrum methodology also improves the skills of the team members, which can lead to better results [14]. The main roles within Scrum are: **Scrum team, Master, Product Owner**, which collaborate during the whole process.

The Master tries to help the team to overcome challenges.
The Scrum team is related to the development, testing and other activities needed.

One of the most essential things used in Scrum is a sprint [14]. A sprint typically lasts between one week and one month. During a sprint the team members work on a task, that was given to them before the sprint. Once a sprint starts the task of each team member cannot be changed until the beginning of the next sprint. At the end of each period, the team has to show an usable product. After each sprint, there is a review of it. The documentation of the sprint, the tasks during it and requirements that should be fulfilled in it are written in the sprint backlog [14]. All tasks that need to be completed are written in the product backlog, where the priority of the tasks can be changed and also new tasks can be added [5]. These tasks and requirements are defined by the product owner and are called user stories. The product backlog is split up in sprint backlogs. The next step is called sprint planning. In this step, the team discusses how to complete each sprint. In the Daily Scrum the team members discuss what they have achieved during the current day. Other parts of the Scrum methodology are: **Sprint Burndown, Product Burndown and Release Burndown charts**, that show the progress of the project [14].

Now that we described the Scrum methodology we will give some disadvantages of it in data-driven projects. In the experiment conducted among students by Jeffrey S. Saltz, Ivan Shamshurin, Kevin Crowston [8] they found, that there was not enough time for understanding the requirements and the data that has been already provided. Therefore, this methodology is not appropriate for overcoming of the unclear requirements. The team members had problems with deciding what to do in a sprint and there were also changes of the tasks during it, which is not allowed by the definition of the methodology. One of the main reasons for that is the exploratory nature of such projects. Because of that, the teams cannot make a right assessment how much time they will need to do the tasks. This leads to the failure of the methodology to deal with the plannability problems. According to the experiment, this methodology had the worst results. The experiment conducted by Jeffrey S. Saltz, Ivan Shamshurin [5] in an organization with many employees confirms the same problems that the methodology had in the previous experiment. For example, the problem with plannability also emerged there. Some of the team members said that it needs to be more flexible during sprints, because sometimes problems emerge that need to be fixed immediately. Therefore, in these teams there were difficulties to finish the task on time. Another finding of the experiment was that an increase in the number of team members lead to reduced quality of work and the daily meeting became less useful. That suggests that teamwork within the team is decreased. Therefore, it is also not appropriate for overcoming of the problems with teamwork and communication [5].
From the disadvantages we can conclude that the Scrum methodology does not help for solving any of the challenges which we defined.

Following, a comparison of the above-described methodologies will be performed. Agile Kanban helps for the improvement of all of the challenges we mentioned. It together with CRISP-DM is more useful for the challenges related to unclear requirements, teamwork and communication problems. However, CRISP-DM is not as flexible in comparison to Agile Kanban.According to our challenges the Scrum methodology had the worst results out of the three methodologies. As we can see from both experiments Scrum has achieved the lowest results, which supports our point.

Now that we discussed some methodologies that are used in practice and problems they bring in a data-driven project environment we will describe some emerging approaches, that are an adjustment or combination of the above-mentioned approaches.

4.2.4 Team Data Science Process

Team Data Science Process (also TDSP) is an emerging approach and was defined by Microsoft in 2016. It is described as an agile and iterative methodology, based on CRISP-DM but also containing elements of Scrum such as backlog artifacts, sprints and roles. The four roles within a TDSP project are: **Group manager, Team lead, Project lead and Individual team contributor**. The process consists of the following 5 phases: **Business understanding, Data acquisition and understanding, Modeling, Deployment and Customer acceptance**. [9]
The phases:
Business understanding: similar to CRISP-DM in this stage the task of the team is to define the project goals and requirements. Another part of this phase is the identification of data sources, that the client needs.
Data acquisition and understanding: in this step the team tries to process the data. Then, it assess whether it is sufficient to complete the goal. The team also attempts to extract new data or to update the current one.
Modeling: in this phase the team creates a model of the data and decides whether the model has good quality.
Deployment: the team deploys the model.
Customer acceptance: in this phase the customer decides whether the model and its deployment fulfill the requirements. [15]

The four roles in the TDSP approach are:
Group Manager: the main job of this role is to create the team or teams according to the standard of Team Data Science Project and to help for the improvement of the teamwork.
Team Lead: this role is responsible for only one team and just like the Group Manager tries to improve the collabora-

tion in it.
Project Lead: this role is responsible for the management of the task that each team member needs to do during the day.
Individual Contributor: this role represents the team members that work on the assigned tasks in the project. [15]
The design of TDSP is very similar to the design of CRISP-DM. Business Understanding and Deployment are defined just like in CRISP-DM. Modeling has the same name as in CRISP-DM but it is a combination of Modeling and Evaluation stage in CRISP-DM. Data Acquisition and Understanding are mixture of Data Preparation and Data Understanding stages.

The main advantage of TDSP over CRISP-DM is that it defines four roles and their tasks during each stage of a project [9]. Another positive characteristic of this methodology is that it leads to better teamwork and collaboration. Its application in a data-driven project can help with the teamwork and communication problems challenge [15] [26]. This methodology cannot help for the solving of the plannability challenge, because the planning sprints have fixed lengths. This leads to a decrease in flexibility [26].

4.2.5 Knowledge Discovery in Data Science

Knowledge Discovery in Data Science (also KDDS) is another approach consisting of 4 phases: **assess, architect, build and improve** [9].

Assess: during this step, the team looks for possible alternatives and assesses them.
Architect: searching for a solution that fulfills the assigned requirements.
Build: in this step, the team handles the development, testing and deployment of the solution.
Improve: the team tries to improve the system. For example to make it faster and more accurate. [16]

KDDS has 5 process stages: **plan, collect, curate, analyze and act**. [9]

Plan: planning the coordination between different organizations that work on this project. Another task is the assessment of the expenses. One more task is the consideration whether the project fulfills the privacy regulations.
Collect: The team has to choose what type of database they are going to use, to achieve big volume and high velocity. Another task is that the teams from different organizations negotiate on ways to provide the other teams with the needed data.
Curate: In this stage, the teams try to understand the data. After that, the teams check whether the expected results will fulfill the privacy and security requirements. In this step the team checks the distributed repositories for curation and analysis. Another task is the consideration of the data quality. One more task is consideration whether sampling is needed.
Analyze: During the first iteration the hypothesis for the data set is made. In the second iteration it is approved or denied. Another task in this stage is that the teams try to find simpler ways to achieve the same business goal. The team needs to choose an algorithm that can run parallel with a huge quantity of data. People working on the project also need to consider concurrency. The teams also need to consider latency, because the analytics of nodes may not be independent. Furthermore, the teams need to analyse whether all communication requirements are fulfilled for each device.
Act: In this last stage the team tries to visualize the data in a way that it is useful and understandable for people who do not have a deep knowledge in statistics. The teams have the task to check whether they fulfill all the data privacy and security requirements when exchanging data between the different organizations working on the project. [16]

In this methodology, the plan stage helps for the coordination of different organizations that participated in the project. The collect stage considers the big volume and the high velocity that such projects can have.
Some studies have shown that KDDS has shown good results in real world applications [9] . Another reason why KDDS can be better than CRISP-DM in a data-driven project environment is because KDDS adds more activities that are directly connected to data-driven projects. For example, it considers the big volume and high velocity of the data that such projects need to handle. This consideration contributes to a better quality of results. Another advantage is that the KDDS methodology helps with coordination between the organizations working on a project that might help for the solving of the teamwork and communication problems [16].

4.2.6 Refined Scrum-DS

Another approach is the Refined Scrum-DS described by Jeroen Baijens, Remko Helms and Deniz Iren [17]. This methodology combines Scrum and CRISP-DM in order to overcome some of the main challenges that come with Scrum. The Refined Scrum-DS takes five of the six stages of CRISP-DM: **Business Understanding, Data Understanding, Data Preparation, Modelling and Evaluation** which are defined in the same way like within CRISP-DM. [17]

In the following we will discuss the artefacts, events and roles within Refined Scrum-DS.
One of the artefacts are the **user stories**. They are defined like in Scrum and describe the tasks. All roles must be present when they are created and they need to be added to the product backlog. They need to be done after the first two steps of this methodology.
The artefact **product backlog** can bring a better understanding of the project. It should be prioritized and created after the first two steps just like the user stories.
The artefact **sprint backlog** should contain data preparation and modelling activities. The Scrum Master and the development team decide which user story from the product backlog needs to be included in sprint backlog and how long each of the user stories will take. It should be created after the first two steps.
The last artefact is the **incremental**. It can help for better understanding of the project or can be a data-driven product, delivered after a sprint and it shows the progress. [17]

The events within the Refined Scrum-DS methodology are: **Sprint, Daily stand-up, Sprint review, Sprint retrospective, Refinement, Sprint zero**.
The **sprint** is just like the sprint in Scrum, but in Scrum-DS a four week sprint is preferred and happens during the

Modelling phase.

Daily stand-up is similar to the daily Scrum and is executed in the data preparation and modelling phases.
Sprint review is a meeting where the result of a sprint is shown to the customer and takes place before the Sprint Retrospective in the Evaluation phase.
Sprint retrospective is a meeting between the team members, in which the team discusses the last sprint. It is done during the Evaluation phase.
Refinement is a meeting about new user stories and their priority assignment. After that the user stories are combined. This leads to the creation of the product and sprint backlog. The Refinement is done within the Business and Data understanding phases.
Sprint zero is sprint before the beginning of the implementation. It is used to find the business goal and the requirements. This is executed in the Data Preparation phase. [17]

CRISP-DM step	Business understanding	Data understanding	Data preparation	Modeling	Evaluation
Events	Refinement		Sprint zero Daily stand-up	Sprint Daily stand-up	Sprint retrospective Sprint review
Artefacts	User stories Product backlog Sprint backlog			Increment	
Roles	Product Owner Scrum Master Development Team		Scrum Master Development Team	Scrum Master Development Team	Development Team Product Owner Scrum Master

Figure 2. The phases, artefacts, events and roles of Refined Scrum-DS [17]

The roles within the Refined Scrum-DS methodology are: **Product Owner, Scrum Master and Development team**. They are defined very similarly to Scrum, which we described earlier [17].

Now that we described Refined Scrum-DS, we will briefly talk about the theoretical advantages it brings. However, the methodology has not been tested and can hide some disadvantages of Scrum.
This methodology is an integration of Scrum into CRISP-DM and improves a lot of its aspects. The design of the methodology adds the sprint zero so that the team can understand the business goal and requirements better. This together with the refinement event helps for defining the user stories better. That is why this methodology can help for the unclear requirements challenge. Furthermore, this methodology should help for the preparation before a sprint. In this methodology, the daily stand-up helps the team to have an effective communication and increases the effectiveness of the sprint events. Apart from that, the scrum master role also helps teams have a better communication. The product owner helps with the communication between the customer and the developers. These roles and events contribute to the better teamwork and collaboration in the project. Because of that, this methodology can handle the teamwork and communication problems challenge. We will mention two other advantages of this approach that help for overcoming the plannability challenge. The first one is that the complexity of the user stories is considered and it is simpler for the team to decide how long each task will take. The second one is that the teams can react better to changes, because there are more interactions between the team members and they get regular feedback from the customer[17].

Here, we summarize the three methodologies and compare them to each other as well as to the other three used in practice. The three methodologies try to be an improvement to the existing ones. TDSP is an adjustment of the CRISP-DM so that it fits better to the new challenges that data-driven projects bring. It handles the teamwork and communication challenge. KDDS is another uprgade of the CRISP-DM. It can also deal with the teamwork and communication challenge. Refined Scrum-DS is an integration of the Scrum into CRISP-DM, because it helps solving the above-described problems of Scrum.

The methodologies used in data-driven projects have not only advantages, but also disadvantages. The approaches we found solve some of the disadvantages, but there is a need for more research because they have not yet been thoroughly tested in a real world environment. One thing that experts have found for sure, is that the methodology that should be used in data-driven projects should be an agile approach[7].

4.3 Approach for fulfilling the 4 V's

As we mentioned in the challenges section there is no standardized lifecycle development process for big data projects, but the IEEE organization has provided a development process for general software projects. The authors of the paper The Design of a Software Engineering Lifecycle Process for Big Data Projects [18] base their new approach on the definition given by the IEEE organization.

Variety is a challenge for big data projects. The approach in the paper tries to overcome it. The term variety is used to describe if the data is structured, unstructured or semi-structured as well as the organization of the data. To achieve variety, the team can look for relevant parts from the data. Moreover, looking in the unstructured data brings the best results, but this method hides a lot of risks. Variety needs to be considered when creating a standard development process for big data projects. [18][23]

Variety is the reason data innovation needs to be included in the business goal. It can bring additional value from the data. Because of its uncertainty, only twenty percent of the goals of the project need to be from data innovation. It is used to look for data trends using different views, ranges, properties, dimensions and also mathematical methods as statistics and multivariable methods. There are three different parts of data innovation. The first is that development team and the customer should discuss about its importance and decide what percentage of the results will be for the data innovation. The second is that, data innovation can be added to data process when needed. The third is that, the cycle steps through which data innovation iterate, are created. [18]

The proposed lifecycle process consists of the characteristic data variety, of the concept data innovation and of the processes software engineering and data analysis. The lifecycle process defined in the paper The Design of a Software Engineering Lifecycle Process for Big Data Projects [18] tries to solve the problems caused by variety by using the following processes:
Data value, result and innovation process: that is where the scope of the data is discussed.

Domain specialist resource management process: this is a process in where the different resources are considered.
Data inventory process: The available data gets a description with all the needed characteristics.
Data requirement analysis process: This is done to assess the expected results and their value.
Data cleaning process: In this process the data gets cleaned at the end of the data innovation process to preserve data variety. [18]
The proposed lifecycle process gives the following advices to overcome challenges that come with data innovation:
Data value, result, and innovation process: this process can be used for the discussion of the changes that need to be made.
Data innovation process: this is where data innovation is executed.
Data processes represent another challenge for which this lifecycle process gives the following recommendations:
Data automation and monitoring process: technical process that aims to create automatic collection and monitoring of data.
Data visualization: this technical process aims to visualize the results, because it is very essential for big data projects.
Data decision support process: it is the last recommended technical process that deals with the assessment of the results whether they are useful for the customer. [18]

In the design of their proposed lifecycle process the authors have combined the model for general software projects by IEEE organization and the processes we described.

Agreement processes	Project processes	Data processes	Technical processes
Data value, result, and innovation process	Project planning process	Data collecting process	Stakeholder requirement definition process
Acquisition process	Project assessment and control process	Data inventory process	Requirement analysis process
Supply process	Decision management process	Data requirement analysis process	Architectural design process
	Risk management process	Data integration process	Data automation and monitoring process
	Configuration management process	Data verification process	Data visualization process
	Information management process	Data analysis process	Data decision support process
	Measurement process	Data modeling process	Implementation process
Organizational project-enabling process		Data simulation process	Integration process
Lifecycle mode management process		Data prediction process	Verification process
Infrastructure management process		Data innovation process	Transition process
Project portfolio management process		Data validation process	Validation process
Domain specialist resource management process		Data cleaning process	Operation process
Human resource management process		Data maintenance process	Maintenance process
Quality management process			Disposal process

Figure 3. The process proposed by the authors of the paper [18]

This lifecycle tries to handle one of the characteristics of big data namely Variety. It causes problems because the data that the teams should process comes from different sources and is in different formats. The data as mentioned above can be structured, unstructured or semi-structured. This is a challenge for the team to process the data. The processes Data value, result and innovation process, Domain specialist resource management process, Data inventory process, Data requirement analysis process, Data cleaning process that we explained are a proposal of the author. Data innovation emerges because of data variety and has to be handled in such projects. Data innovation is included in the business goal, because it brings more value to the results but it is very uncertain. Therefore, it is recommended that the team should not invest all of its efforts in it. It should be only twenty percent of the goals of the project. The authors describe the processes: **Data value, result, and innovation process, Data innovation process, Data processes** that can solve the challenges related to the data innovation. The proposed lifecycle development process is a good foundation for the future work on establishment of a standard for such a process in big data projects. This process needs to be tested in practice so the advantages and disadvantages it hides can be discovered.

5. DISCUSSION

Now we will discuss the different approaches we found and describe which challenge they could help solving.
The unclear requirements challenge can be overcome by Agile Kanban methodology and Refined Scrum-DS. In Agile Kanban the preparation step and the Kanban board help for better understanding of the business goals and the requirements of the customer. In Refined Scrum-DS the refinement event helps for claification of the requirements.
Agile Kanban, TDSP, KDDS and Refined Scrum-DS can help with the teamwork and communication problems in data-driven projects. The Kanban board in Agile Kanban improves the communication [2]. The artefacts, defined in the TDSP methodology can lead to improved communication in the team [15]. Because of the Plan stage, the KDDS methodology can also help with communication and collaboration between organizations working on the same project. Lastly, the Refined Scrum-DS methodology can have better communication between team members because of the scrum master role and the daily stand-up. The product owner role can also lead to better communication between the stakeholders and the team.
The challenge about the plannability can be solved by Agile Kanban and Refined Scrum-DS. The Agile Kanban methodology can help for solving this challenge, because of its flexibility [2]. As mentioned earlier, the Refined Scrum-DS methodology can lead to higher flexibility, because the team members get feedback more frequently.
The last challenge we talked about was fulfilling the 4 V's. The approach we mentioned needs to be tested and can serve as groundwork for the definition of a standardized life-cycle development process.

Now we will briefly talk about some challenges that we did not find solutions for. The first that we mentioned is the lack of clear definition of roles within a data-driven project team. That is because the field is constantly evolving and a lot of different skills are needed, which makes it difficult to define unified roles. We never found a concrete definition of the roles, which is further supported by the papers of Jeffrey S. Saltz, Nancy W. Grady [4]. In their work we found that there are currently different organizations defining different roles, which either have the same name but different job during a project or there are some roles that overlap but have a different name. This leads to more confusion among the employers when they try to find employees with the needed skills for a project. The roles defined above can serve as groundwork for the further definition of unified roles. The last challenge is the lack of life-cycle process that is used in practice. Because of that, many big data projects fail or cannot complete the business goals assigned by the customer.

6. CONCLUSION

In conclusion, we would like to summarize our paper and also talk about what needs to be done in the future for a higher success rate of data-driven projects.

The main problems we found in the literature were the lack of standard for the lifecycle development process, process methodology and definition of the roles of team members in such projects. The reason, standards from other fields do not work in a data-driven environment is that they do not consider their characteristics. For example, projects involving big data need to consider Volume, Velocity, Veracity and Variety, which cause additional requirements. We found some approaches for these challenges, that can be theoretically good solutions. Nevertheless, they need additional research and testing in a real world applications, because they can hide some drawbacks. One of the main objectives we concluded is that there is a big need for unification and standardization of the above-mentioned roles, which will lead to a much better success rate of these projects. The difficulty of solving these challenges comes from the rapidly evolving field and the many different types of data-driven projects.

Data-driven projects have become more important and popular which makes adjustment to their requirements more and more mandatory.

7. REFERENCES

[1] Ali Sever - *M*odeling Distributed Agile Software Development for Big Data Projects: Evolution in Process.

[2] Jeffrey S. Saltz, Ivan Shamshurin - *A*chieving Agile Big Data Science: The Evolution of a Team's Agile Process Methodology.

[3] Jeffrey S. Saltz, Ivan Shamshurin - *B*ig Data Team Process Methodologies: A Literature Review and the Identification of Key Factors for a Project's Success

[4] Jeffrey S. Saltz, Nancy W. Grady - *T*he Ambiguity of Data Science Team Roles and the Need for a Data Science Workforce Framework

[5] Jeffrey S. Saltz, Ivan Shamshurin - *A*chieving Agile Big Data Science: The Evolution of a Team's Agile Process Methodology

[6] Jeffrey S. Saltz, Ivan Shamshurin - *B*ig Data Team Process Methodologies: A Literature Review and the Identification of Key Factors for a Project's Success

[7] Patrícia Franková, Martina Drahošová, Peter Balcoa, - *A*gile project management approach and its use in big data management

[8] Jeffrey S. Saltz, Ivan Shamshurin, Kevin Crowston - *C*omparing Data Science Project Management Methodologies via a Controlled Experiment

[9] Jeffrey Saltz, David Wild, Nicholas Hotz, Kyle Stirling - *E*xploring Project Management Methodologies Used Within Data Science Teams

[10] Piatetsky, G. - *C*RISP-DM, Still the top methodology for analytics, data mining, or data science projects. www.kdnuggets.com/2014/10/crisp-dm-top-methodologyanalytics-data-mining-data-science-projects.html

[11] Jeffrey S. Saltz - *T*he Need for New Processes, Methodologies and Tools to Support Big Data Teams and Improve Big Data Project Effectiveness

[12] Ana Azevedo, Manuel Filipe Santos - *K*DD, SEMMA AND CRISP-DM: A PARALLEL OVERVIEW

[13] Ahmad, M. O., Markkula, J., & Oivo, M. (2013, September). Kanban in software development: A systematic literature review. In Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on (pp. 9-16). IEEE.

[14] Apoorva Srivastava, Sukriti Bhardwaj, Shipra Saraswat - *S*CRUM Model for Agile Methodology

[15] Microsoft - *T*eam Data Science Process - https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/lifecycle-business-understanding

[16] Nancy.W. Grady - *K*DD meets Big Data

[17] Jeroen Baijens, Remko Helms, Deniz Iren - *A*pplying Scrum in Data Science Projects

[18] Yen-Tai Lin, Sun-Jen Huang - *T*he Design of a Software Engineering Lifecycle Process for Big Data Projects

[19] David J. Hand - *P*rinciples of Data Mining

[20] Foster Provost, Tom Fawcett - *D*ata Science and its relationship to Big Data and Data-Driven Decision Making

[21] Jing Gao, Andy Koronios, Sven Selle - *T*owards A Process View on Critical Success Factors in Big Data Analytics Projects

[22] VB Staff *h*ttps://venturebeat.com/2019/07/19/why-do-87-of-data-science-projects-never-make-it-into-production/

[23] Ryohei Fujimaki *h*ttps://www.datanami.com/2020/10/01/most-data-science-projects-fail-but-yours-doesnt-have-to/

[24] Pervaiz Akhtar, Jedrzej George Frynas, Kamel Mellahi and Subhan Ullah *B*ig Data-Savvy Teams' Skills, Big Data-Driven Actions and Business Performance

[25] Fernando Martinez-Plumed, Lidia Contreras-Ochando, Cesar Ferri, Jose Hernandez-Orallo, Meelis Kull, Nicolas Lachiche, Maria Jose Ramirez-Quintana and Peter Flach *C*RISP-DM Twenty Years Later:From Data Mining Processes to Data Science Trajectories

[26] *W*hat is TDSP? Team Data Science Process https://www.datascience-pm.com/tdsp/

All of the figures we used are from the referenced papers

A Systematic Study and Collation of Amazon AWS and Microsoft Azure Cloud Architecture Design Patterns

Pavan Nadkarni
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
pavan.nadkarni@rwth-aachen.de

Sehrish Kahn
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
sehrish.kahn@rwth-aachen.de

Alex Sabau
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
alex.sabau@swc.rwth-aachen.de

ABSTRACT

Since their establishment, design patterns have been an important means to support the development of software architectures and applications. They have become the de facto standard when it comes to designing software architectures and code. Over the last decades' technology and the internet have evolved rapidly. This has changed the way we use, approach, and perceive software. As a result, requirements such as scalability, availability, and distribution are being significantly prioritized during the development and deployment of software applications. Cloud computing is being used to address them. This area is dominated by the hyperscalers such as Amazon, Microsoft, and Google. They define and maintain their own collection of design patterns to assist customers utilizing their specific cloud platforms. This has led to a vast patchwork of design patterns being used in the context of cloud computing. While there exists de facto standardization of design patterns in architectural and software development domains like object-oriented software, there is a clear deficit in its adoption in the cloud computing domain. In this paper, we study cloud design patterns published by the two hyperscalers, Amazon and Microsoft, to identify redundancies between them. We use their knowledge and expertise to create an overview of design patterns defined by them in the field of cloud computing. We then provide a methodology for categorizing these patterns into groups and apply standardization techniques to certain groups to obtain a homogeneous representation of patterns and then discuss the results obtained.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.11 [**Software Engineering**]: Software Architectures—*patterns, domain-specific architectures*

Keywords

Cloud Computing, Hyperscalers, Design Patterns, Cloud Design Patterns, Cloud Architecture Patterns

1. INTRODUCTION

Looking into the history, it was in 1948 when the first software program was successfully stored electronically and executed on a computer [12]. Since then, software applications have transformed and revolutionized world we live in. Software application development practices from then on has evolved continuously. They are being standardized to constantly improve properties like flexibility, reusability, reliability, and operational and performance efficiencies of software [4]. Most of the time, obstacles and challenges encountered during software development were alike across specific domains. Soon the group of four computer scientists and engineers', Erich Gamma et al., famously known as the Gang of Four (GoF) [11] provided generic solutions to address such challenges in the context of object-oriented software development and termed them as design patterns. They published the book "Design Patterns: Elements of Reusable Object-Oriented Software" which defines 23 such design patterns [11] which soon became the de facto standard collection of design patterns when developing object-oriented software.

With the widespread adoption of technology, billions of devices connected over the internet are performing computations and exchanging data in real-time. The data generated by these components has grown exponentially over the past decade. This has given rise to the need for on-demand distributed and remote computational and storage capabilities. Cloud computing gained wide popularity when companies like Amazon, Microsoft, and Google, foreseeing the potential in this sector, decided to offer such on-demand services to consumers and enterprises by leveraging their existing investments in data centers [14].

The rapid ongoing shift in the adoption of cloud services for application development has introduced various new challenges such as isolation, distribution, elasticity, automated management, and loose coupling [6]. To overcome these challenges, each hyperscaler has defined and documented a catalog of design patterns that cater to their customers in developing cloud applications on their platform. This has led to a lack of standardization in such pattern definition, often resulting in patterns with different names performing the same or similar functionality.

Christoph Fehling in his dissertation [6] talks about the necessity of pattern organization and its homogeneous representation in the field of cloud computing. In this paper, we hypothesize that there exist several redundancies between design patterns and their definitions and aim to find answer to the following question: "Does Amazon AWS and Microsoft Azure have cloud design patterns that address the same or similar issue but are named differently?". To answer this, we present a systematic approach to identify and organize such patterns by aggregating those defined by Amazon AWS and Microsoft Azure and generalizing them by decoupling them from the individual cloud providers' context.

The paper is structured as follows. First, the theoretical background of design patterns, cloud computing, cloud native applications, and hyperscalers are formally introduced in Section 2 to the reader to establish a common terminology. Notably, various categorizations of cloud computing, the importance of design patterns in application development, the important cloud application properties, and the necessity for adopting a standardization process in defining cloud computing design patterns are introduced to the reader. In Section 3, we provide a brief overview of the related work. Subsequently, in Section 4, we present our methodology for categorizing various design patterns offered by the two hyperscalers, Amazon AWS and Microsoft Azure. In Section 5, we present an overview of our results obtained after applying the methodology outlined in Section 4 and then discuss the results in Section 6. In the end, in Section 7, we draw a conclusion and present future work that can extend the results.

2. BACKGROUND

2.1 Design Patterns

As quoted by Christopher Alexander in his book "A Pattern Language", "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [1]. Although the quote concerns about those patterns found in towns and buildings, it is nevertheless similar when it comes to design patterns in software development, where both try to find a solution to a problem in a context.

As defined by Erich Gamma et al. [11], a design pattern consists of the following essential components:

- ***Pattern Name***
- ***Problem***
- ***Solution***
- ***Consequences***

2.2 Cloud Computing

The National Institute of Standards and Technology (NIST) defines cloud computing as "A model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction" [16]. Additionally, it categorizes the cloud model into five essential characteristics, three service models, and four deployment models [16]. Here we just describe the three service models that help to relate the applicability of a design pattern to a service model.

2.2.1 Service Models

- ***Software as a Service (SaaS)***: In this service model, the consumer uses the provider's applications running on a cloud infrastructure accessible from various client devices through either a client interface or a program interface. The consumer has limited control over user-specific application configuration settings and does not manage or control the underlying cloud infrastructure or even the individual application capabilities.

- ***Platform as a Service (PaaS)***: This capability provides the consumer the ability to deploy custom-created or acquired applications onto the cloud infrastructure. Applications like these can be developed using programming languages, libraries, services, and tools supported by the provider. The consumer can manage and control only the deployed applications and possibly configuration settings for the application-hosting environment and not the underlying cloud infrastructure.

- ***Infrastructure as a Service (IaaS)***: This service model allows the consumer to provision processing, storage, networks, and other fundamental computing resources to deploy and run arbitrary software. This can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure. But has control over the operating systems, storage, deployed applications, and limited control of networking components.

2.3 Cloud Native Applications

Applications that are built to be deployed on the cloud are termed as cloud-native applications. Such applications need to exhibit certain inherent properties to benefit from the cloud environment and its offerings. The IDEAL properties as proposed by Christoph Fehling [6], help applications to efficiently handle the workload.

2.3.1 Properties

- ***Isolated State***: This property aims at isolating the handling of state information of an application to a minimal number of its components and preferably utilize the data storage functionality offered by the cloud provider. This improves the scalability and resiliency of applications.

- ***Distribution***: Cloud environments are typically large distributed systems. Cloud-native applications deployed on such environments are generally decomposed into multiple components whose functionality is largely distributed to effectively utilize multiple resources in the cloud environment.

- ***Elasticity***: The cloud-native applications should have the ability to acquire and relinquish required resources automatically during runtime depending on the workload changes without affecting the user of the application. This includes dynamic scale-out and scale-in of the application instances.

- ***Automated Management***: This property allows cloud-native applications to appropriately react to dynamic changes in resource demands and failures during runtime in an automated mechanism without the requirement for any human interaction.

- ***Loose Coupling***: Loose coupling is a well-established concept in distributed applications. Minimizing dependencies between application components makes them independent and improves their scalability and resiliency factor. This eases provisioning and decommissioning of tasks and also reduces the impact of failing application components.

2.4 Hyperscalers

Hyperscalers are technology giant companies like Amazon, Microsoft, Google, and many others that dominate in the cloud computing domain. These companies offer an extensive catalog of specialized cloud platforms and services to the public and enterprises to build, host, and maintain their software applications.

This research paper aims to compare patterns defined by Amazon AWS and Microsoft Azure. It does not scope in the views of other hyperscalers as only the aforementioned companies provided a publicly advertised design patterns catalog.

3. RELATED WORK

As a starting point of our research, we considered "Cloud Design Patterns" by Microsoft [13] and Amazon's official documentation "Amazon AWS Prescriptive Guidance" as our references. The majority of the topics described by Alex Homer et al., [13] are applicable to all distributed systems and for almost all cloud platforms. Microsoft Azure and Amazon AWS have provided common solutions to recurring problems in the cloud environment by using pattern-based description format [13].

However, according to our knowledge and research, there is significant work done by Christoph Fehling, Johanna Barzen et al. [7] in the analysis, application, and categorization of cloud design patterns and by Christoph Fehling [6] regarding architectural and management principles but not particularly in identifying redundancies among design patterns of different cloud providers.

The knowledge shared by Christoph Fehling, Thilo Ewald et al. in [8] is somewhat similar to our research on the systematic study of cloud design patterns. Here they discuss and provide the solution for pattern identification in cloud computing that is offered by different cloud providers, following different pattern approaches. To help developers in determining which cloud design pattern is suitable in a particular scenario, an elaborated pattern format in the form of "pattern authoring toolkit" [9] is provided to users for traceable recognition of cloud design patterns [8]. In contrast to the pattern identification based on its formatting and documentation style, we categorized design patterns based on the functionality they offer to cloud users. Christoph Fehling, Thilo Ewald et al. propose in [8], a well-defined format for structuring and organizing design patterns by providing generic graphical notations and a specific pattern template. This helps users in better management of design patterns regardless of the cloud provider. Whereas, on the other hand, we grouped similar cloud design patterns that offer the same solution to a recurring problem for better identification of design patterns between AWS and Azure.

More generic pattern identification and authorization are discussed by Christoph Fehling, Johanna Barzen et al. in [7] under various domains that involve multiple industry partners. In the cloud computing domain, the research done by Christoph Fehling, Johanna Barzen et al. in [7] has categorized cloud environment by defining different deployment options and different service models with the help of different design patterns that are interrelated by pattern language. "A pattern language is a set of related patterns that act as building blocks in that they can be carried out in one or more pattern application sequences where each subsequent pattern builds upon the former" [5]. Our research is specific to the cloud domain and we took inspiration to use cloud service models to categorize interrelated design patterns instead of categorizing the cloud environment.

In the field of cloud computing, patterns are strengthening the design and development of software applications. However, in industries, design patterns are still used in a very casual way, especially at the time of decision making. In the conference paper, Christoph Fehling, Johanna Barzen et al. [7] talk about procedures and techniques to categorize cloud environment on "a conceptual level by reviewing existing applications developed for cloud and those that should be migrated to it". Our categorization of the design patterns adds knowledge to pre-existing information that is published by the two cloud providers Amazon AWS and Microsoft Azure.

Steve Strauch et al. in [17] and Christoph Fehling, Frank Leymann et al. in [10] have covered pattern categorization for migrating application data to the cloud. Their research is specific to the categorization of cloud design patterns that deal with data. However, in our research, we considered design patterns belonging to various categories and not just data. Furthermore, the need for categorizing patterns that provide a solution to problems of the same nature and domain has been highlighted by Christoph Fehling, Johanna Barzen et al. in [7].

When hyperscalers define solutions using design patterns, they tend to couple their platform-specific offerings into the solution to assist their customers' software development process. As mentioned by Christoph Fehling, Thilo Ewald et al., "the industry-driven evolution of cloud computing tends to obfuscate the solutions to common underlying problems that are provided by different hyperscalers" [8]. This was an important motivation for us in our research to provide common abstract pattern names for design patterns of Microsoft Azure and Amazon AWS that eventually serve the same purpose.

To the best of our knowledge, there has been no study conducted to date to identify redundant or similar cloud design patterns between the two hyperscalers, Microsoft Azure and Amazon AWS. Therefore, we primarily rely on the knowledge and expertise of the two hyperscalers, Amazon and Microsoft [13] to identify redundancies and standards among their patterns.

4. METHODOLOGY FOR CATEGORIZATION

This is a qualitative and descriptive research to categorize existing cloud design patterns offered by the two hyperscalers Microsoft Azure and Amazon AWS.

The need for abstract guidelines in the form of design patterns to develop software solutions arose with the evolution of cloud computing. As this evolution was mostly industry-driven, cloud providers have offered inconsistent guidelines to underlying problems by following different formats, terminologies, and approaches. The use of provider-specific terminologies thus makes it unclear for cloud users to gain provider-independent knowledge. This research paper is tailored to eliminate the underlying issue and is oriented towards comparing the cloud design patterns of the mentioned hyperscalers to identify redundancies between them.

To find interrelations between cloud design patterns, we use pattern-oriented analysis based on the definitions and documentation provided by Microsoft Azure and Amazon AWS to extend the existing work of Christoph Fehling [6]. The research is performed in the following three phases:

1. Grouping of design patterns based on the definitions provided by hyperscalers.
2. Assigning generic names to each group of similar patterns.
3. Assigning pattern categories to grouped patterns.

Next, we discuss each of the three phases in detail.

4.1 Grouping of Design Patterns

The need to group cloud design patterns of Microsoft Azure and Amazon AWS was to capture common concepts and abstract away from the models and implementations of both cloud providers. Our grouping gives cloud users a quick overview of design patterns that address the same problem. This makes pattern identification very straightforward.

We formulated the grouping strategy for cloud design patterns by introducing three different types of groups.

- ***Identical Pattern Group***: This is the most refined yet basic group. The identical pattern group will contain all cloud design patterns of Microsoft Azure and Amazon AWS that match not only in their definitions but also in their names. While comparing cloud design patterns descriptions, we considered the possibility that these patterns were explained using provider-specific terminologies, but they essentially address similar issues.
- ***Similar Pattern Group***: The comparison criteria for the similar pattern group get a bit relaxed in terms of the pattern name. This group includes those design patterns that offer the same solution to the same recurring problem, but they differ in their names and may incorporate certain provider-specific implementation details. The description of patterns by Microsoft Azure and Amazon AWS is mostly provided by using provider-specific formats and terminologies. The names given to many patterns consider the context where a cloud provider tries to explain a problem faced by them. This leads to ambiguity in naming when mapping has to be made between patterns that address similar problems across different cloud providers. Thus this group draws the comparison based on a problem situation and its proposed solution.
- ***Distinct Pattern Group***: This group is composed of those cloud design patterns that are exclusive to Microsoft Azure or Amazon AWS. For such patterns, the problem and the solution described by one cloud provider are not offered by the other. No pattern mapping is possible here as cloud design patterns contained in this group are independent. Thus for this group, we captured design patterns as two lists, one for each hyperscaler.

4.2 Generic Names for Interrelated Design Patterns

Christoph Fehling, Johanna Barzen et al. in their definition of pattern language, stated that "Patterns are interrelated through references in order to guide users during their application. Through these interrelations, the patterns of a certain domain form a so-called pattern language" [7].

We found interrelation among design patterns by performing the grouping in phase one; then, in order to give a quick insight and guide the cloud users during their application, we proposed generic and intuitive pattern names for each set of the interlinked design patterns. Generic naming gives the actual feasibility of mapping the design pattern of Microsoft Azure and Amazon AWS. During the process, we analyzed the characteristics of both cloud providers' design patterns and proposed generic pattern names and solutions. These design pattern names and solutions are standardized as they are devoid of any provider-specific terminologies.

Generic names were assigned to patterns in the similar pattern group as this group consisted of those patterns that addressed the same issues but differed in their names. While most patterns in the identical pattern group matched in their definitions and also in their names, there existed few patterns that differed in their names by a word or two. We also assigned generic names for such patterns.

4.3 Pattern Categorization

This segment focuses on the identification and exploitation of shared characteristics defined for cloud design patterns of the hyperscalers Microsoft Azure and Amazon AWS. These characteristics enable cloud users to swiftly list design patterns under a particular category and to identify those that fulfill the same requirement. To assess the equivalence between cloud design patterns, we used the following pattern categories [13] that are defined by the hyperscalers.

- ***Modern Application Design***
- ***Messaging Integration***
- ***Operational Excellence***
- ***Data Management***
- ***Performance Efficiency***
- ***Reliability***
- ***Security***

The categories represent different domains where the design pattern's application enhances certain architectural, functional, or non-functional aspects of the cloud application's properties. This emphasizes the fact that design patterns under the same category address solution to the same problem. It additionally helped in cross-examining our grouping strategy based on names and pattern definitions provided by the hyperscalers Amazon and Microsoft.

5. OVERVIEW OF FINDINGS

As part of our research, we methodically analyzed 94 design patterns published by the hyperscalers Amazon and Microsoft for their cloud platforms. We categorized them into the 3 groups as detailed in Section 4.1. After categorization, we identified 11 design patterns under the identical pattern group that are similar in all aspects in Table 1.

Among the remaining 83 patterns, with our meticulous and iterative classification process, we identified 14 design patterns that fall into the similar pattern group. These design patterns address similar issues but were inherently categorized and named differently by the hyperscalers. We standardized such patterns by decoupling them from the cloud provider's reference and proposed a generic name as captured in Table 2.

As the catalog of design patterns considered is very vast and due to spatial limitations, it is extremely difficult to present a short and conclusive discussion of each design pattern. Thus we present only the generic problem and solution for the design patterns in the identical and similar pattern groups in Sections 5.1 and 5.2. We categorized design patterns that are exclusive to the hyperscalers under the distinct pattern group. Here we do not present the specific pattern definition provided by Amazon [2] [3] and Microsoft [15] and also the list of distinct patterns. These are captured in a separate resource file that is hosted on GitLab.[1]

5.1 Identical Pattern Group

For each pattern in Sections 5.1 and 5.2, we discuss the generic problem, which describes the problematic situation in a cloud application, during when the pattern's application should be considered. For this generic problem, we then present a generic solution that describes how the application of the pattern resolves the issue. The problems and solutions presented in these sections are termed generic since they are free of hyperscaler specific terminologies.

- ***Bulkhead Architectures***
  - ***Generic Problem***: A cloud-based application may include multiple services. Each service might have one or more consumers. Excessive load or failure in a service will impact all consumers of the service.
  - ***Generic Solution***: The Bulkhead Architecture pattern resolves this issue by partitioning service instances into different groups, based on consumer load and availability requirements. This design helps in isolating failures, and provide continued functionality for a subset of consumers, even during a failure.
- ***Cache-Aside***
  - ***Generic Problem***: Applications generally use caching patterns to improve their performance. This can lead to data inconsistencies in cache and data store.
  - ***Generic Solution***: The Cache-Aside pattern caches data on demand into a dedicated cache memory to provide faster access to data. This improves the application's performance and helps to maintain data consistency between cache and the data store.
- ***Choreography***
  - ***Generic Problem***: Often applications use the orchestrator pattern to reduce point-to-point communication between services. This has some drawbacks as it introduces tight coupling between the orchestrator and other services that participate in the processing of the business transaction. The addition and removal of new services would require making changes in the communication path between the orchestrator and the service, which can be complex and hard to maintain.
  - ***Generic Solution***: The choreography pattern, allows each service to decide the processing strategy of an operation instead of relying on a centralized orchestrator for communication. Since there is no point-to-point communication anymore, this pattern helps reduce coupling between services. Additionally, it eliminates the performance bottleneck caused by the orchestrator.
- ***Circuit Breaker***
  - ***Generic Problem***: In a distributed environment, calls to remote resources and services can fail due to transient faults. However, there can also be situations where faults are due to unanticipated events and can be long-lasting. In such situations, it would be optimal to allow the operation to fail immediately to prevent unnecessary blockage of critical system resources by the failing requests and attempt to invoke the service only if it is likely to succeed.
  - ***Generic Solution***: The Circuit Breaker pattern helps in preventing an application from repeatedly trying to execute an operation that is likely to fail. It acts as a proxy with different states for operations that can fail. It monitors the number of recent failures that have occurred and then uses this information to decide whether to proceed with the operation or return an exception immediately by switching to an appropriate state.
- ***Command Query Responsibility Segregation (CQRS)***
  - ***Generic Problem***: Applications can have read and write workloads that are often asymmetrical, with different performance and scale requirements. Developing data access models for such systems which address the representations, parallel operations, performance, and management of security and permissions of the read and write operations on the data can become complex.
  - ***Generic Solution***: The CQRS pattern separates the read and update workloads for a data store. Commands are used to update data while queries perform the read operation. This helps to detach the workload between command and query part based on the requirements for throughput, latency or consistency and allow the application

[1]Link to the Gitlab document: https://git.rwth-aachen.de/pavanmn/fullscalesoftwareengineeringseminar.git

Table 1: Identical Pattern Group

Pattern Category	**AWS Pattern Name** [a]	**Azure Pattern Name**	**Generic Pattern Name**
Reliability	Bulkhead Architecture	Bulkhead	Bulkhead Architecture
Data Management, Performance Efficiency	Cache-Aside	Cache-Aside	Cache-Aside
Modern Application Design, Performance Efficiency	Choreography	Choreography	Choreography
Reliability	Circuit Breaker	Circuit Breaker	Circuit Breaker
Modern Application Design, Data Management, Performance Efficiency	CQRS	CQRS	CQRS
Modern Application Design, Data Management, Performance Efficiency	Event Sourcing	Event Sourcing	Event Sourcing
Modern Application Design, Operational Excellence, Reliability	Leader Election	Leader Election	Leader Election
Messaging Integration, Performance Efficiency	*Priority Queue **	Priority Queue	Priority Queue
Messaging Integration, Operational Excellence	Pub/Sub	Publisher/Subscriber	Publisher/Subscriber
Modern Application Design, Data Management, Reliability	Saga Distributed Transactions	Saga	Saga
Modern Application Design, Operational Excellence	Sidecar	Sidecar	Sidecar

[a]The patterns that are italicized and marked with asterisk symbol refer to the list of old Amazon patterns. Since these design patterns are still endorsed by Amazon they were considered within the scope of our research work.

to improve its scalability, performance, and security.

- ***Event Sourcing***
  - ***Generic Problem***: Many applications work with data by maintaining only the current state of the data and making updates on it when required using the traditional CRUD approach. This approach can limit the ability to scale, create update conflicts during concurrent access, and fails to capture the history of operations on the data item.
  - ***Generic Solution***: The Event Sourcing pattern uses an append-only store to capture all operations performed on a data item instead of storing just the current state of the data. This helps to reconstruct the application's state at any point in time by reprocessing the operation history. This improves the scalability, performance, and consistency of applications and prevents the need to perform data synchronization.
- ***Leader Election***
  - ***Generic Problem***: Cloud applications perform multiple tasks in a coordinated manner and access resources that are shared. When working with shared resources, it becomes critical to avoid conflicts and interferences with other tasks accessing the shared resources.
  - ***Generic Solution***: The Leader Election pattern allows electing an instance as the leader among the coordinating instances that undertakes the responsibility of managing coordination among other instances in a distributed application. This can help in preventing conflicts and interference among instances while accessing shared resources.
- ***Priority Queue***
  - ***Generic Problem***: In cloud applications, messaging queues are typically used to delegate tasks to other services for background processing. As the volume of requests increases, it becomes necessary for the application to prioritize specific requests such that these should be processed earlier than lower priority ones.
  - ***Generic Solution***: The priority queue pattern uses a highly reliable queue to prioritize and maintain requests sent to various services. It allows the application to process requests with higher priority quickly than those with lower priority. Multiple queues can be created depending on the varying priority levels. This helps in developing highly reliable applications.
- ***Publisher/Subscriber***
  - ***Generic Problem***: In cloud applications, components need to communicate and exchange information with other components as new events oc-

cur. This makes it essential to establish an asynchronous messaging system that allows the exchange of information with interested consumers in an efficient manner.

  - ***Generic Solution***: The Publisher/Subscriber pattern helps in establishing an asynchronous medium of communication among the different services without coupling the sender and the receiver. The subscribers can subscribe to various channels for which they would like to be notified. Whenever a new event gets generated for a channel the publisher notifies all consumers that have subscribed to the particular channel.

- ***Saga***
  - ***Generic Problem***: Distributed cloud applications require a way to coordinate transactions between multiple services to maintain data consistency. Cross-service data consistency requires a cross-service transaction management strategy and a method to roll back the entire transaction in case of failure.
  - ***Generic Solution***: The Saga pattern offers a failure management strategy to preserve data consistency across microservices in a distributed transaction scenario. A saga is a sequence of transactions that updates each service and publishes a message/event that helps in triggering the next transaction step. The subsequent transaction is executed based on the event's outcome and in case of failure, the saga triggers a compensating transaction to roll back the prior transactions.

- ***Sidecar***
  - ***Generic Problem***: Applications generally require peripheral tasks which implement functionality such as logging, monitoring, and configuration. These functionalities are typically integrated into the main application and tightly coupled to it. This creates interdependence among the core functionality and the peripheral task. Any failure in the peripheral tasks can lead to an outage of the application's core functionality.
  - ***Generic Solution***: The Sidecar pattern allows to bundle and launch a helper container alongside the main container that offers the core service. The helper container shares the same lifecycle as the parent application and provides supporting features to it. This pattern also helps to decouple the application's core service from other peripheral functionality.

5.2 Similar Pattern Group

- ***API Gateway***
  - ***Generic Problem***: When a client consumes multiple services, setting up multiple communication endpoints for each service is a challenging and tedious task. Additionally, the client not only needs to be aware of all the endpoints but also needs to adapt to any endpoint changes or service refactoring done on the server side.
  - ***Generic Solution***: The API Gateway pattern resolves this issue by introducing a gateway component between the client and the backend services that expose their API. The gateway receives the request from the client and performs a reverse proxy to route the request to the necessary internal service. In this way the client is abstracted from the numerous internal endpoints and is remains unaffected if these services are refactored.

- ***External Configuration Store***
  - ***Generic Problem***: Applications runtime environments utilize configuration information typically held in files. In a cloud-hosted scenario, an application with multiple instances has configurations specific to each instance. This makes it a challenge to manage changes to local configurations across multiple running instances of the application. It also limits the possibility of sharing common configuration settings across multiple applications.
  - ***Generic Solution***: The External Configuration Store pattern enables storing the configuration data in a centralized store and detaches it from the application deployment package. This facilitates for easier management of configuration data. It also allows multiple applications and their instances to easily access this shared configuration data during startup.

- ***Service Callback***
  - ***Generic Problem***: Cloud applications generally rely on API calls over the HTTP(S) protocol and follow REST semantics to expose various services. At times these services can execute tasks in the backend that may be long-running. This makes it infeasible for clients to wait for a reply from the backend in cases of a synchronous request-reply pattern.
  - ***Generic Solution***: The Service Callback pattern allows the clients to make a synchronous API call in a non-blocking way using the asynchronous poll model. This gives the appearance of asynchronous processing. This can be implemented using an HTTP poll model and configuring a status endpoint. The client can invoke the status endpoint to know the status of the resource/task. This builds a loosely coupled architecture avoiding bottlenecks caused by synchronous communication.

- ***Static Content Hosting***
  - ***Generic Problem***: Web applications typically include static contents like HTML pages, images and videos which are to be hosted. Rendering these static contents takes up computing resources that can alternatively be used for other purposes.
  - ***Generic Solution***: The Static Content Hosting pattern allows hosting the static contents of a web application directly on cloud-based storage offered by the cloud provider that can directly deliver contents to clients. This avoids the usage of

Table 2: Similar Pattern Group

Pattern Category	AWS Pattern Name [a]	Azure Pattern Name	Generic Pattern Name
Modern Application Design, Operational Excellence	API Gateway	Gateway Routing	API Gateway
Modern Application Design, Operational Excellence	*Bootstrap **	External Configuration Store	External Configuration Store
Messaging Integration, Operational Excellence	Decouple Messaging	Asynchronous Request-Reply	Service Callback
Modern Application Design, Data Management, Performance Efficiency	*Direct Hosting **	Static Content Hosting	Static Content Hosting
Data Management, Performance Efficiency	*Inmemory DB Cache **	Cache-Aside	Cache-Aside
Modern Application Design, Operational Excellence, Reliability	Monitoring Integration	Health Endpoint Monitoring	Application Monitoring
Modern Application Design, Operational Excellence	*Multi Load Balancer **	Gateway Offloading	Shared Functionality Delegation
Modern Application Design, Data Management, Security	*Private Distribution **	Valet Key	Token-Based Resource Access
Messaging Integration, Reliability, Performance Efficiency	*Queuing Chain **	Queue Based Load Levelling	Queue Based Load Levelling
Reliability	Retries and Backoff	Retries	Retry
Data Management, Performance Efficiency	*Sharding Write **	Sharding	Sharding
Reliability, Performance Efficiency	*Stamp **	Deployment Stamp	Stamp
Data Management, Performance Efficiency	*Storage Index **	Index Table	Index-based Access
Messaging Integration, Data Management, Performance Efficiency	*Web Storage **	Claim-Check	Reference-Based Messaging

[a]The patterns that are italicized and marked with asterisk symbol refer to the list of old Amazon patterns. Since these design patterns are still endorsed by Amazon they were considered within the scope of our research work.

computational resources for rendering static content and saves costs.

- ***Application Monitoring***
  - ***Generic Problem***: With applications hosted remotely in the cloud, it becomes necessary to monitor these applications to verify if these services are available and working correctly. Monitoring the cloud-hosted services is challenging than monitoring on-premises services because one may not have full control of the hosting environment.
  - ***Generic Solution***: The Application Monitoring pattern allows incorporating functional checks in an application to monitor it. These checks can be exposed externally using published API endpoints to allow users to monitor different functional metrics of the application.
- ***Shared Functionality Delegation***
  - ***Generic Problem***: Applications generally require shared/specialized functionality that are common to many other applications like logging, monitoring, authentication, and authorization. Configuring, managing, and maintaining these shared or specialized services deployed with every application increases administrative overhead and the possibility of making an error during configuration.
  - ***Generic Solution***: The Shared Functionality Delegation pattern allows delegating the configuration of certain shared/specialized functionalities components like load balancers and gateways that are common to multiple applications. This also reduces the administrative overhead of configuration and maintenance for multiple deployments.
- ***Token-Based Resource Access***
  - ***Generic Problem***: Cloud applications typically use data stores to handle the upload and download of data directly without requiring the application to perform any processing nor use valuable resources such as compute, memory, and bandwidth. But this requires applications to securely control access to the data in a granular way to allow untrusted clients to communicate directly

with the data store to perform the required read or write operations.

  - ***Generic Solution***: The Token-Based Resource Access pattern lets applications provide clients with a limited time-restricted direct access to a specific resource by issuing tokens like URLs. This offloads the data transfer functionality from the application and helps to minimize cost and maximize scalability and performance.

- ***Queue-Based Load Levelling***
  - ***Generic Problem***: Cloud applications are composed of multiple services that are subjected to varying amounts of loads depending on the number of dependent services. Predicting the volume of requests to a service when it is used by a number of tasks running concurrently is very difficult. Peaks in demand can cause an overload and can also result in the failure of the service.
  - ***Generic Solution***: The Queue-Based Load Levelling pattern uses a queue acting as a buffer between a task and a service it invokes. This can help to ease out the load and minimize the impact of heavy loads on availability and responsiveness for both the task and the service. This additionally enables asynchronous and loose coupling of services.

- ***Retry***
  - ***Generic Problem***: Transient faults are quite common when applications interact with services deployed in the cloud. These issues typically get self-resolved and return successful results when retriggered after a suitable delay.
  - ***Generic Solution***: The Retry pattern allows applications to handle ephemeral failures when trying to communicate with services and resources. The application can perform a retry operation during such failures. This makes the applications are more resilient, reliable, and fault-tolerant.

- ***Sharding***
  - ***Generic Problem***: Data stores hosted by a single server can have limitations concerning storage space, computing resources, and network bandwidth. While scaling up resources of the server helps to resolve issues momentarily, it is generally not a long-term resolution for commercial cloud applications that need to support large numbers of users and high volumes of data.
  - ***Generic Solution***: The Sharding pattern involves horizontal partitioning of the data store in units known as shards. The data persisted into the data store is distributed onto different shards based on certain pre-identified columns called keys. This technique improves the scalability and performance of an application when storing and accessing huge volumes of data.

- ***Stamp***
  - ***Generic Problem***: The work of setting up the operating system, middleware, and applications required for any virtual server takes considerable time, effort, and expense. Deploying multiple such server instances to address performance and scalability issues would require the setup work to be performed multiple times for each instance making this a tedious and time-consuming process.
  - ***Generic Solution***: The Stamp pattern allows the creation of a machine image that already has the required environment like the operating system, middleware, and applications set up as part of the image. This image can then be used to swiftly deploy multiple independent copies of the environment known as stamp and improve the scalability of your solution.

- ***Index-Based Access***
  - ***Generic Problem***: Applications typically execute a large number of queries on the data store to fetch data. Searching for a specific set of data from a huge data store can be time-consuming and cause delays in response if the storage or indexing of data is not optimized.
  - ***Generic Solution***: The Index-Based Access pattern creates indexes over fields in the data store that are frequently accessed by the application queries. These indexes act as metadata and allow applications to locate the data swiftly and improve the overall searching performance.

- ***Reference-Based Messaging***
  - ***Generic Problem***: A messaging-based architecture should have the ability to send and receive large files/messages like images, documents, and multimedia. But sending them over the message bus directly consumes high resources and bandwidth and can also slow it down. Additionally, messaging platforms impose restrictions on the size of messages that can be exchanged.
  - ***Generic Solution***: The Reference-Based Messaging pattern allows applications to deliver large files to clients by storing such files in a data store offered by the cloud provider and then sending a reference for it via a messaging system. Clients interested in processing the specific message obtain the reference to retrieve the data from the data store.

6. DISCUSSION

Our study on design patterns published by Amazon and Microsoft supports our initial hypothesis that there exist redundant design patterns defined by the two hyperscalers. We identified that there exists an overlap in the pattern definition among the hyperscalers, but this accounts for only 11 percent of our research data. Based on the results presented in Section 5, we see that the overall overlap is significantly higher for the similar pattern group as compared to the identical pattern group. After applying our standardization process, the percentage of common patterns between the hyperscalers Amazon and Microsoft has increased by nearly 15 percent. This data contributes to a clearer understanding of the need to adopt standardization techniques in the design pattern definition process. It also backs the statement citing the need of defining a pattern language in the field of cloud computing for pattern organization and its homogeneous representation as stated by Christoph Fehling in his dissertation [6].

Here we acknowledge that the generalizability of our results is limited to the design patterns defined by Amazon AWS and Microsoft Azure. The results do not take into account design patterns defined by other available hyperscalers like Google and RedHat. This was well beyond the scope of our research due to the non-availability of a published list of design patterns adopted by these hyperscalers.

7. CONCLUSION AND FUTURE WORK

In our paper, we collected and reviewed various cloud design patterns published to date by the hyperscalers Amazon and Microsoft for their respective cloud platforms. We categorized these design patterns into 3 different groups based on their definition and sample use cases provided by the hyperscalers. While this method was not entirely objective, we relied upon the expertise of the hyperscalers and various other research works to base our claims. In instances where decisions had to be taken subjectively, we referenced various available resources to gain necessary knowledge and understanding of a design pattern and its use case. The knowledge gained was accompanied by multiple iterations of classification and peer reviews before stating the results.

With extensive research, we discovered that a fraction of design patterns coincide among the hyperscalers. With this result, it can be stated that while there exist no standards defined for pattern definition in cloud computing, there are some patterns that have been widely adopted and have thus become the de facto standard set of design patterns in the domain of cloud computing. Applying our standardization process to the remaining patterns helped in extending the list of common design patterns between the two hyperscalers, Amazon and Microsoft. The results assist anyone researching in the field of cloud computing to understand such design patterns and their applications in a generic way. It is also very beneficial for enterprises that are looking to migrate their applications between the two hyperscalers. The results also help us to understand how the adoption of standardizations benefits capturing knowledge more efficiently and reduces redundancy and ambiguity.

Our research work can be extended further by considering different methodologies or properties or design pattern definitions of other hyperscalers like Google and RedHat if available for categorizing and grouping the design patterns. For example, deduce a mapping function using certain fundamental cloud application design properties that groups design patterns into certain pre-defined categories. Since cloud technology is evolving rapidly, new patterns get defined over time. A similar methodology can be adopted to standardize the new patterns and append them to the existing catalog of identified cloud computing design patterns. The results also raise new questions like "Why do the pattern definitions and names provided by the hyperscalers differ for the design patterns captured in the similar pattern group?", "Why some categories of patterns like application migration are captured only by a single hyperscaler?". Additionally, tools can be developed for automatically performing pattern suggestion, identification, and categorization using mapping functions that utilize the captured catalog of cloud design patterns as the underlying database.

8. REFERENCES

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press.

[2] Amazon AWS. Aws cloud design patterns. `http://en.clouddesignpattern.org/index.php/Main_Page`.

[3] Amazon AWS. Aws perspective guidance. `https://aws.amazon.com/prescriptive-guidance`.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A., Patterson, A. R. nad Ion Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, 2009.

[5] R. Cope and A. Naserpour. *Cloud Computing Design Patterns.* 2017.

[6] C. Fehling. *Cloud Computing Patterns Identification, Design, and Application.* PhD thesis, January 2015.

[7] C. Fehling, J. Barzen, U. Breitenbücher, and F. Leymann. A process for pattern identification, authoring, and application. pages 1–10. Institute of Architecture of Application Systems, University of Stuttgart, Germany, March 2014.

[8] C. Fehling, T. Ewald, F. Leymann, M. Pauly, J. Rütschlin, and D. Schumm. Capturing cloud computing knowledge and experience in patterns.

[9] C. Fehling, T. Ewald, F. Leymann, M. Pauly, J. Rütschlin, and D. Schumm. Pattern authoring toolkit. `http://cloudcomputingpatterns.org/authoringtoolkit.zip`, 2012.

[10] C. Fehling, F. Leymann, S. T. Ruehl, M. Rudek, and S. Verclas. Service migration patterns - decision support and best practices for the migration of existing service-based applications to cloud environments. 2012.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

[12] T. Haigh, M. Priestley, and C. Rope. *ENIAC in Action: Making and Remaking the Modern Computer.*

[13] A. Homer, J. Sharp, L. Brade, M. Narumoto, and T. Swanson. *Cloud Design Patterns.* February 2014.

[14] ISO/IEC JTC 1/SC 7 Software and systems engineering. Iso/iec 25010:2011 systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models. Technical report, 2011.

[15] Microsoft. Azure cloud design patterns. `https://docs.microsoft.com/en-us/azure/architecture/patterns/`.

[16] National Institute of Standards and Technology. The nist definition of cloud computing. Technical report.

[17] S. Strauch, V. Andrikopoulos, T. Bachmann, and F. Leymann. Migrating application data to the cloud using cloud data patterns. 2012.

Quality of Service Requirements of Workloads in Compute Clusters

Mahta Khoobi
RWTH Aachen University
Aachen, Germany
mahta.khoobi@rwth-aachen.de

Muhammad Daniyal Danish
RWTH Aachen University
Aachen, Germany
daniyal.danish@rwth-aachen.de

ABSTRACT

Computer science is a vast field that is both expanding and changing at an enormous speed. This modification implies on everything within this vast territory of bits, ranging anything from algorithms, to hardware to deployments. For example, serverless architecture has changed the software deployment immensely from what it was a few years ago. Regardless of all the advancements in the sub-domain of infrastructure, there are still various gaps that need to be filled, one of which this paper tries to answer.

In the modern world of software development, enterprise applications are expected to run on clusters rather than a single server. Through managing these clusters by available platforms, we can configure and change the quality requirements expected from the deployed software. For example, to satisfy the availability requirements of a software in Kubernetes, memory and processing power can be configured, but these are not the only configurations to be kept in mind. In the event of excess load from users (Scalability Requirement), an additional resource might be required for constant delivery of services. The Quality of Service (QoS) requirements - which are defined by users' demands, need to be translated into Quality of Service attributes provided by clusters and implemented in the software development phase.

This research paper aims to focus on i) what compute cluster requirements exist? ii) and What relationship exists between QoS requirements of a service/software and computes clusters' configurations? The standard Quality of Service requirements of a Software will be divided into subattributes and mapped to QoS features offered and provided by clusters.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*Infrastructure, Deployment, software configuration management*

Keywords

quality of service requirements, compute clusters, QoS of workloads

1. INTRODUCTION

Ever since computers were invented, there has been constant effort to upgrade both the hardware and the software capabilities. For Software, the efficiency of algorithms keeps on improving. Similarly, more powerful hardware is available, than it was in the year before. This can be seen from the comparison between the number of cores in the supercomputers from 2020 [3] and 2010 [2] in which the cores have increased 30 folds. [27]

As the usage of the internet increased, applications had millions of active users during the peak time. Hence more powerful computers were required to handle the workload [8]. Workloads can be defined as a group of tasks, running applications, or parallel batch jobs on clusters. [20]

Initially, vertical scaling was suggested as a solution to meet hardware requirements. In vertical scaling, the resources like memory and CPU for a single server are added to provide more resources for the application to run. One of the downsides of vertical scaling is that the maximum point for scaling is limited to the size of the server. Although this is still one of the current solutions based on the working environment, this is not the case anymore when it comes to resource utilization, cost reduction and power consumption [29, 36].

To consider these factors, modern-day architecture uses horizontal scaling, in which rather than increasing the hardware within the system, virtualization is used to satisfy the hardware requirements and these systems communicate and transfer data with each other. By virtualization, a single server can be treated as multiple virtual machines, running the workloads [8]. One way of realizing virtualization is the use of containers and clusters, which are one of the fundamental concepts of software development in the current era.

2. BACKGROUND

Clusters and cluster computing have seen a substantial rise in adoption in the last decade. Startups and tech giants alike are leveraging cluster-based architectures to deploy and manage their applications especially when workload increases.[22] But, what is a cluster? And what is the relationship between clusters and containers? And why might one want to consider using a cluster to host their application? To be more clear, some terms will be explained

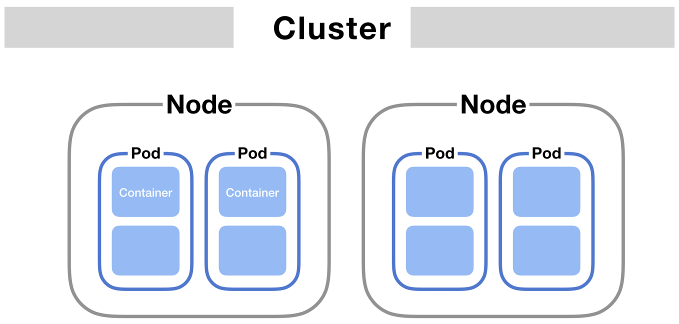


Figure 1: Diagram visualising the relation between containers, pods, and clusters [23]

in the following section, some of which will be used for the rest of this paper.

2.1 Containers

According to the **Figure 1**, the container is the basic part of a cluster, logically inside a pod. Containers hold individual applications and their respective resources, software, and configuration and can be perceived as software encapsulated operating systems' processes, which have their namespace and resource limits, like CPU and memory [14]. Containers help applications to run more efficiently and be transported more quickly. Unlike virtual machines, which are similar in concept, containers do not have their guest operating system; instead, they run dependent on an existing operating system. This gives more flexibility to the system since it can run anywhere. Therefore, containers have enabled agility in development lifecycles [35].

2.2 Pods

Pods are a group of one or more containers, which are always scheduled together (always run on the same machine), with shared storage/network resources, and a specification for how to run the containers. To access a pod, each pod has its IP address and port. Therefore, containers in a pod have the same IP address and port. Pods are a great solution for managing groups of closely related containers that depend on each other and need to co-operate with the same host to accomplish their purpose [33, 6]

2.3 Nodes

Node is like a host for pods and a Pod always runs on a Node. A Node is a worker machine, either a virtual or a physical one. Each node has some components like controlling unit and container runtime (like Docker), which is responsible for pulling the container image and running the application [17, 33].

2.4 Clusters

A cluster is a collection of pods, which hosts storage and networking resources that be used to run various workloads that comprise the system. The entire system may consist of multiple clusters [33]. To visualize the definition, a cluster can be seen as a board that provides the circuitry to run all the pods (which have the container instances in them) in an orchestrated manner as defined by the users [6].

2.5 Deployment

A deployment is a resource object that provides declarative updates to applications. A deployment gives description about an application's life cycle, such as which images to use for the app, the number of pods there should be, and the way in which they should be updated. A deployment ensures running and availability of desired number of pods, the recorded and versioned update process,which can be paused, continued, and rolled back to previous versions. [28]

2.6 Cluster Computing

Clusters work together in the execution of intensive compute and data tasks that would be not feasible via a single computer. Clusters are used mainly for high availability, load-balancing, and scalability purposes. Cluster Computing is highly available as it maintains redundant nodes which are used to provide non-stop service in terms of the system's failure. The performance of the system is improved here because even if one node fails at some point, there is another standby node that will take the responsibility of the task and eliminates the concept of single points of failure without any hindrance [38]. Since multiple computers are linked together in a cluster, they share the computational workload as a single virtual computer. The user's requests are received and distributed among all the standalone computers to form a cluster. This results in balanced computational work among different machines, improving the performance of the system. [31]

To visualize the relationship between containers, pods, and clusters, again take a look at **Figure 1**. In summary, the relation can be defined as below [6]:

- A container runs logically in a pod (though it also uses a container run time)
- A group of pods, related or unrelated, run on a node
- A node is a working machine for the cluster, either virtual or physical
- A cluster can contain many pods, related or unrelated [and] grouped under the tight logical borders called name-spaces

Although clusters and horizontal scaling sound like a brilliant solution, they come with their challenges such as the need for good architecture and manageability. Moreover, it might get slow because the resource allocation in horizontal scaling might cause variable delays in VM deployments. Therefore, a controlling platform is needed to orchestrate all the clusters based on workloads' requirements and resource specifications [36].

2.7 Container Orchestration Platforms

Along with containers concepts, the microservice architectural pattern has been adopted, which encapsulates an individual core application functionalities as microservices and builds the whole system by composing them. These independent components are easier to develop, manage, and (horizontally) scale [12].

These micro-service applications usually consist of thousands of clusters providing services, to satisfy the quality of service requirements of enterprise applications. To benefit from micro-service applications and this powerful pattern, the clusters should be reliable, available, scalable, and secure.

Container orchestration platforms can be defined as a system that provides an enterprise-level framework for managing containers at scale. That means making sure that all the containers that execute various workloads are scheduled to run physical or virtual machines. These platforms simplify container management and provide a blueprint not only for defining initial container deployment but also for managing multiple containers for achieving the quality of service requirements.

Some of the key capabilities of a container orchestration platform are:

- Providing high availability and fault tolerance
- Security
- Simplifying networking
- Ability for continuous deployment and scalability
- Providing monitoring and governance
- Cluster state management and scheduling

There are many container orchestration platforms available today like Docker, Kubernetes, Amazon Web Services (AWS), Microsoft Azure, which can be deployed to manage the clusters. Kubernetes is an open-source platform, which has been produced to automatize the deployment and management of clusters of Docker containers [15]. Docker [9] containerizes individual processes and allows them to be run within a separated container. The service developer can create Docker images containing desired service elements and Kubernetes will deploy and manage the components.

To achieve reliable, available, and scalable software, Kubernetes and other similar platforms give the users the interface to configure the clusters according to the business requirements and budgets [33].

2.8 Quality of Service Requirements

For an enterprise software to be successful, it is very essential to fulfill all the requirements that a customer expects from it. This involves both the expectations from the software and also the expectations from the deployment. No matter how wonderful software is, it would not be deemed useful if it is not available when it is needed. To emphasize more on the importance of this let us take an example of an online video conferencing application called Zoom. By March 2021, Zoom was able to increase its revenue generation by more than 300 percent. People argue that it was because of the COVID-19 pandemic whereas actually, it was because Zoom was able to scale quickly compared to its competitors. Its architecture was able to handle the increase in the load from users and kept on delivering the service. Had it not been a scalable system, then things would have been very different for Zoom now. [16]

From the discussion above, the importance of the customers' requirements for the quality of service can be understood, with which we can have a successful running software. The Quality of Service Requirements (QoS Requirements) is aimed to provide continuous service even during the event of excess load. Therefore, the availability, scalability, security, and maintainability of the service can be fulfilled. Often these requirements are interrelated and certain trade-offs must be considered. For example, to make a system more secure a compromise on performance requirements have to be made. Since there are many configurations that one must consider while designing the deployment architecture, it is possible to skip a few trade-offs or relations. Therefore, one can conclude that if the deployment architecture was not planned properly, things could have been turned out differently, for example, the mentioned software; Zoom. This shows that it is important to first identify these requirements expected by the workloads and then map them to the respective clusters' requirements, which are provided by the current features in clusters' orchestration platforms.

This paper addresses this gap by providing a relation between QoS requirements and their respective compute cluster configurations. In the next section, the paper discusses related work which has been done regarding QoS requirements of both clusters and batch jobs. Further down it writes what major QoS requirements for the workload are and what configurations we can do with the clusters to ensure those QoS requirements will be met. It talks also briefly about what benefits we get from each requirement and gives a more pragmatic view of how clusters can be configured.

3. RELATED WORK

As a starting point for our research, we have considered Kubernetes documentations [17] and books [33, 18] as it is the most powerful cluster management platform, in which the more advanced topics related to the quality of service management have been explained. The majority of technical implementations they have provided are mostly cover reliability, availability, scalability, and security. Moreover, to consider a standard for quality of service requirements of workload, we considered the standard of ISO/IEC 25010 [1]. In this standardization, one quality model is defined to do the quality evaluation of a system. This model determines which quality characteristics will be taken into account when evaluating the properties of a software product. This mainly covered quality of service requirements from stakeholders' perspective, namely functionality, performance, security, maintainability, etc. which categorizes the product quality into characteristics and sub-characteristics.

However, according to the research we have done, significant work has been done regarding applying the quality of service requirements expected from the workload to the clusters' quality of service management. This paper [24] focuses

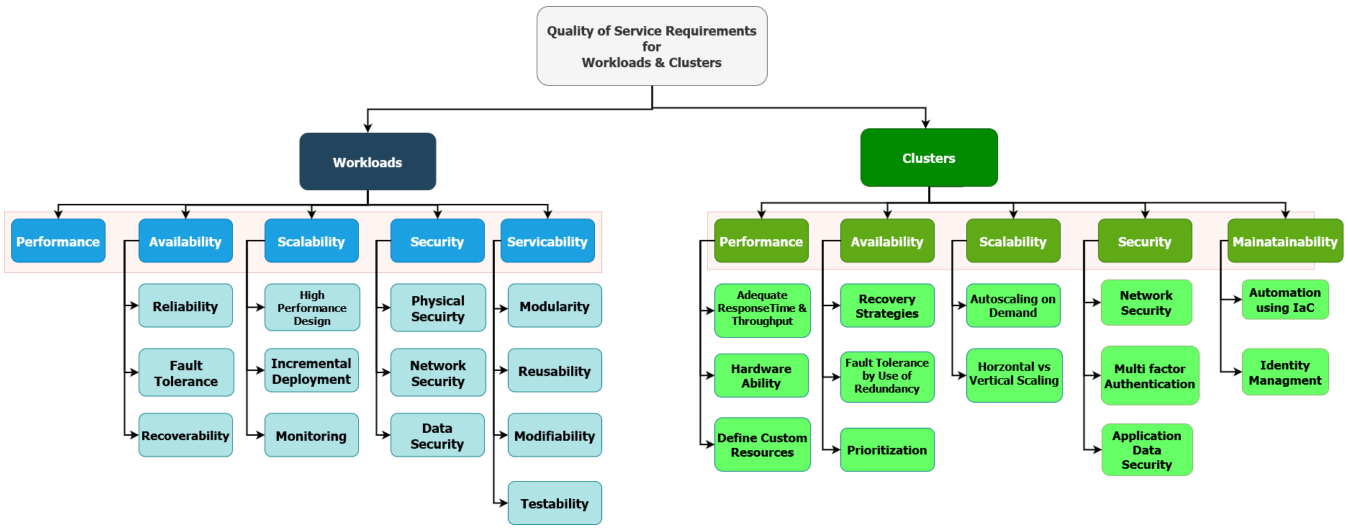


Figure 2: Diagram visualising the overview of quality of service requirements for workloads and clusters and their relations

on a service-oriented computing environment and provides a hybrid method to develop complex applications and satisfy user's personalized requirements. With the fast growth of web services, each user requirement may be supported by a set of services with similar functionality but different QoS. This hybrid method is proposed to solve the QoS-aware service composition. This method uses a search scheme to combine the search space reduction and the exploration of the reduced search space. It handles QoS-aware service composition optimization effectively and efficiently.

To achieve QoS requirements relating to resource utilization and scaling, a generic system is proposed [39] to both dynamically scale up the Kubernetes clusters to guarantee QoS and at the same time avoid wasting resource utilization while providing required QoS. The proposed system contains four modules: monitor module, QoS module, scaling module, and executing module. First, the monitor module uses open-source tools to monitor and store the real-time status of a Kubernetes cluster. Then, to guarantee QoS, the QoS module has a method to automatically decide a threshold for CPU utilization that can meet users' requirements. Next, the scaling module provides a scaling algorithm to get an ideal number of nodes in the Kubernetes cluster. Finally, according to the ideal number of nodes, the executing module adjusts the scale of the Kubernetes clusters to run the application. As a result, The proposed system improves CPU utilization of a Kubernetes cluster by 28.99%.

One of the important problems with cloud resource management is resource provisioning which is necessary to be handled during the workload fluctuations in cloud-based applications. Resource provisioning is a concept which deals with the resource allocations for services and makes sure that cloud services do not run out of hardware resources under any circumstances [30]. Resource provisioning depends on the number of users and the number of resources as important factors and it can automatically scale up based on the clusters workloads and QoS metrics provided by the users. However, since the submitted cloud workloads by users are heterogeneous in terms of QoS metrics, therefore, their analysis and identification to meet QoS requirements agreed in service level objectives (SLOs) can play an important role to provision the cloud resources in a cloud environment. This paper [10] proposes an efficient solution based on a metaheuristic-based clustering mechanism to analyze the cloud workloads. The proposed approach utilized a combination of the genetic algorithm and fuzzy C-means technique [37] for clustering the heterogeneous cloud workloads based on QoS metrics.

4. QUALITY OF SERVICE REQUIREMENTS FOR WORKLOADS AND CLUSTERS

Quality of Service requirements is technical requirements of a system that features attributes like performance, scalability, availability, and serviceability. They are usually identified by the business needs and hence differ from project to project. For example, if the business needs to be available 24 hours a day, throughout the year, then the QoS requirement for availability must be addressed. Similarly, if the system has requirements in which it can face peak hours with immense load from users, then scalability requirements should be addressed.

QoS requirements are closely interrelated. Satisfying one requirement/ system quality might hinder the satisfaction of the second requirement/ system quality. One such pair can be performance and security, or availability and serviceability. For example, adding higher levels of security in a system will affect the performance in a negative way. This in turn will affect the availability. To improve the availability one can add additional servers, which will affect serviceability - maintenance cost. Understanding these trade-offs and how system qualities are interrelated is a key to design a successful system. This system should satisfy both business requirements and business constraints.

To have an overview of the quality of service requirements in both workloads and clusters, you can see the diagram (**Figure 2**). Each requirement in the workloads branch has been divided into sub-requirements and each requirement in the clusters branch has been divided into different strategies for the realization of the requirement with the aid of cluster management platforms and other environmental settings.

The section below further talks about the system qualities which affect deployment design along with some further guidance on which factors should be considered while formulating QoS requirements.

4.1 QoS requirements of Service

4.1.1 Performance

Performance is one of the business requirements which is generally expressed in non-technical terms. For example, a user for a web-based system might describe the performance as something: User expects a reasonable response time typically no greater than 2 seconds.

Considering these business requirements, one must analyze all the use cases and see how the requirements can be addressed in a better way. Usually, in some cases, load conditions are added to see how the system performs under load. After testing on these requirements one can write performance in terms of response time, and for load conditions, it can be then written as the sum of throughput and response time. Since it is almost impossible to guarantee no errors, therefore, it is always recommended to write down the number of allowable errors. Following are the possible two ways in which this requirement for performance can be written:

- Response time of web pages should not take longer than 2 seconds throughout the day, sampled at every 10 seconds intervals with 6 errors per million.
- During defined peak periods of the day, the system must be able to register at least 16 new users in less than 15 seconds with less than 6 errors per million.

Usually, Performance requirements are closely related to the availability requirements (failover impact) and latent capacity (capacity available to handle unusual peaks), and therefore, it is important to distinguish between these requirements. Like availability, the performance also has a relation with security. In most of the cases, performance goes down when the system is being tried to make more secure [11]. Otherwise, it is extremely important to provide security as much as possible especially for some applications like Banking and online transactions, it is better to find a trade-off between these qualities, so the system can have a bit of both in the right quantities. With this in mind, consumer trust can be achieved which is undeniable requirements expected from online banking [5].

4.1.2 Availability

Availability is measured to specify the uptime of the system. It is usually measured in the percentage of times that the system is accessible for users. The time when the system is not available for the users (downtime) could be because of several reasons like software, hardware, network, or power loss. However, many times systems are shut down for maintenance or upgrades, which is called scheduled downtime and this is not considered as downtime. Availability's measurements can be formulated as an equation with which we can calculate the uptime of the system as follows:

$$\text{availability} = \text{uptime}/\left(uptime + downtime\right) \cdot 100$$

Normally, the availability is measured in terms of nines **(see Table 1)** and it defines the amount of time (per year) for an available system with running services [7]. We can also consider the converse view which shows the unexpected downtime of a system. Therefore, an average system with 99.9 percent availability has availability of three nines, which means 364.64 days available time during a year and 8.76 hours of downtime. It is straightforward that adding additional nines can affect the deployment designs significantly. The table below depicts the downtime in terms of hours concerning the number of nines of availability for a system that runs throughout the year every day (8760 hours).

Fault-Tolerant Systems: Fault-tolerant systems are systems that can continue to give their services even during hardware or software failures. Availability requirements of four or more nines usually describe fault-tolerant systems. Fault tolerance can be achieved by providing redundancy in both software and hardware in terms of CPUs, network devices and memory [21]. For example, in hardware redundancy, a server can be made fault-tolerant by using an identical server running in parallel, with the same operations for the backup server. As a redundancy approach to the software, all client information in a database can be replicated to another database in case of failure of the main one [26].

For a system to be fault-tolerant, either all single points of failure must be eliminated or tackled. A single point of failure is a component or service which is part of the critical path of the system and can not be backed up by redundant systems. The system will fail if one of these components fails and therefore it is very important to identify all single point of failure components during designing systems. It is both costly and difficult to implement fault-tolerant systems. Therefore, it is recommended to understand the user requirements properly to see if it is important to implement a fault-tolerant system in the first place.

Prioritizing Services Availability: From a user's point of view, usually few services are more important than others, therefore, availability is normally applied to service rather than a system as a whole. For example, a service that deals with real-time communication would have little to no significance on the overall availability of a system. However, a service on which several other services rely on such a directory service would have a greater impact on the system as a whole. Therefore, these services should be given greater priority when considering the availability of a system. We can divide services into the following categories to identify their importance:

- **Mission critical:** Always available systems, such as database services.
- **Must be available:** Services which should be available. It is okay to have reduced performance such as messaging service.
- **Can be postponed:** Services that must be available in a given time frame such as calendar services for internal company services.
- **Optional:** Good to have services. For example, in some environments, chatbots can be ignored.

To choose which strategy we should pick is dependent on what expectations users have from the service and whether the scope we are working on is highly available demanding like online transactions or less care about availability in online chatbot of a retails shop.

Table 1: Availability in "Nines"

Number of nines	Available Percentage	Uptime per year	Downtime per year
2	99%	361.35 days	3.65 days
3	99.9%	364.64 day	8.76 hours
5	99.999%	~365 days	5.26 minutes

Loss of Service: Availability includes scenarios when the service is no longer present. The design and requirement must include this possibility and narrate whether the connection should be restarted or if there is a way to resume the service from the point it was interrupted. It should also narrate how the system will react to this loss and what deployment steps should be taken to minimize this loss.

4.1.3 Scalability:

Scalability is the ability to add up resources to the existing resources to satisfy the additional load from users. Scalability typically requires the addition of hardware resources but it should not demand any kind of change in the existing architecture of the deployment nor does this should require downtime because of the addition of resources. Scalability should not be done in a way that takes away the service from the user and leads to a bad user experience instead.

Like with availability requirements, scalability implies more on individual services rather than the system as a whole. However, for the components/services on which many other components/services depend, such as Directory service, scalability can have a system-wide impact.

Scalability for a system on a deployment design level is not specified specifically unless the requirement document mentions these scalability requirements and needs. However, the deployment architecture should include some tolerance for unexpected cases for which scalability should be done. For such cases, the critical components which should be scaled as well as the percentage increase in the software, hardware, and network should be specified in the QoS requirement guide.

Growth Estimation: For a system to be able to stand against the sudden increase in workload, it is very important that growth estimation has been done during deployment architecture design. This involves extrapolation of project workload over time, evaluation of use cases under which a sudden increase of load from users can be expected, and hunches that might never get filled. There are three keys that help to develop scalability requirements for scalable systems:

- **High-Performance Design Strategy:** This strategy allows to absorb growth and do better scheduling. It requires an increased budget for availability requirements and it assumes that the workload increases over time.
- **Incremental Deployment:** Clear milestones for the systems upgrade are defined. These milestones are usually load-based requirements.
- **Extensive Performance Monitoring:** The system is being monitored all the time to see if there is an increase in load in which case extra resources are being added.

4.1.4 Security:

Security requirements are considered complex requirements because they can involve all deployment stages. Mainly, in security analysis, we identify critical services and threats for those services and how they can affect the organization as a whole. Following are few things that should be considered while implementing security:

- **Physical Security:** Physical security means providing security to any type of hardware that your system is using. It can be servers, data rooms, routers, etc. This is most important because other types of security will mean nothing if anyone can go and damage the hardware.
- **Network Security:** This includes implementing strategies to avoid unauthorized access, denial of service (DOS) attacks, and tempering on your network which can be done via ports, firewalls, access control lists, etc.
- **Application and Application Data Security:** This includes using all the practices in development that help avoid data leakage and expose any security threats.

4.1.5 Serviceability:

It is a measure of the efficiency with which the deployed system can be maintained, changed, or upgraded. It includes subattributes like modularity, reusability, modifiability, and testability. Modular systems or services are the ones that are flexible to change and the change would not affect other systems/services. Reusable components or services can be used in other systems, and modifiable services are flexible to an efficient change without decreasing the quality of the current system. To measure serviceability, testability helps to define and perform test criteria for a service [1].

4.2 QoS configurations/features cluster offers:

Different systems need different quality of service requirements. How to design and implement the right system depends on those requirements. As it is mentioned earlier, to implement the workload requirements in clusters, some portable and extendable platforms for managing containerized workloads and services have emerged that facilitate both declarative configuration and automation as well as the quality of service requirements. Clusters' orchestration platforms like Kubernetes, provide different strategies for each quality of service requirements based on the system's conditions, which will be discussed in detail in this section;

4.2.1 Scalability:

There are different strategies to implement scalability in clusters which can be either software-based or hardware-based [18].

- Ability to autoscale on demand:

Clusters orchestration systems can watch the clusters and when the CPU or other metrics pass over a threshold, then they increase the capacity of resources automatically. By defining the autoscaling details, both minimum and maximum range of scaling. For example in Kubernetes, Without the maximum limit, Kubernetes will keep creating pods–groups of containers in clusters– until all resources are exhausted and via cloud environment with autoscaling of VMs then we will incur a significant cost and if there is no minimum limit, then all pods could be terminated. This is why it is important to consider the upper threshold to reduce any extraneous costs.

- Performance monitoring of the system to determine when to autoscale:

  We may encounter resource scarcity while we horizontally scale the clusters. Therefore, there are some factors like DaemonSets and stateful sets, division of workload in a namespace, etc. which will affect deciding about efficient resource scalability.

- Available resources to add to cluster:

  Via horizontal autoscaling, we can scale the cluster, storage, and other objects. However, in the end, we're limited by the physical or virtual resources available to our cluster. If all the clusters are running at their full capacity, the cluster management platform will just fail to scale. On the other hand, in dynamic workloads, scaling down the clusters is possible, but with static scaling (via Hardware) you will still pay for the excess capacity. One solution to this problem is to make use of storage solutions provided either by Cluster Orchestration Platforms or the Cloud Storage Solutions. Moreover, understanding the fluctuations of the workloads and considering the cost/benefit ratio of having excess capacity versus having reduced response time or processing ability can help significantly.

4.2.2 Availability:

There are many aspects to highly available clusters, such as ensuring that the control plane, which manages worker nodes and pods in clusters, can keep functioning in the face of failures, protecting the cluster state in storage components, protecting the system's data, and recovering capacity and/or performance quickly. More aspects to availability can be defined as below:

- Maximize uptime of systems by recovery strategies:

  The level of the immediate recovery varies depending on the workload's requirements. For some systems which are not sensitive to downtime, the Best Effort strategy will work, which means; in case of downtime, there is no predefined strategy like redundancy, etc. However, for more sensitive systems there are some quick recovery and self-healing strategies that are provided by most of the cluster orchestration platforms. To maximize uptime of the system, different strategies can be deployed based on different occasions [33].

  Hot Swapping: By replacing the failed component on the fly without/with minimum interruption to the users, the availability of the system can be achieved. In this method, the stateless components can be replaced without harm by only redirecting the clients to the new component. However, for stateful components, systems will either refuse to have in-flight transactions because the respective users are resilient to failures or the systems will keep a hot replica in sync, which may cause system overhead and a low performance if it is not partially implemented for the system.

  Leader Election: In the case of having many worker components and a master component, it is possible to make the workers redundant and swap the master component by electing a new leader. This election can be done following the Hot Swapping method. Some master components, such as the scheduler, can't have redundant instances active at the same time. Otherwise, it will be chaos that different schedulers handle pods and components. The correct way is to have these components run in leader election mode. This means that multiple instances are running, but only one is active at a time and if it fails, another one is elected as leader and takes its place.

  Smart Load Balancing: Load balancing is about distributing the workload across multiple components that service incoming requests. In case of failure of some components, the load balancer should stop sending workload to those components and be provided by fresh components. Kubernetes have provided this ability through services and labels.

- Fault tolerance by use of redundancy in hardware and data Level:

  If a critical component fails and you want the system to keep running, you must have a copy of the component ready to go. In some conditions, for example for stateless pods, the cluster orchestration platform may provide replication controllers or replica sets to provide redundancy. However, some components like the master component or storage component need also to get backed up by a redundant component and storage, respectively, to provide fault tolerance and avoid data loss. Moreover, the API servers, which are part of clusters, can be redundant and available to the users. Since the API servers are mostly stateless, there is no need to coordinate them but a load balancer may be deployed to distribute the workload to each.

- Prioritizing service availability:

  The scheduler can prioritize the group of components, namely pods, and if a high-priority pod could not be scheduled, the scheduler will find a lower-priority pod and evict it to continue with the scheduling. In this way, Pods which are prioritized to be available will be on the top list for scheduling. One major security problem with this method is that In an unsafe cluster environment with unknown users, a malicious user could create high-priority Pods, causing other Pods to be evicted/not get scheduled. An administrator can use different strategies to avoid that. For example, in Kubernetes, admins can deploy ResourceQuota to prevent users from creating pods at high priorities. Disaster Recovery Environment

4.2.3 Security:

Clusters are exposed to an increasing number of attacks coming from public networks. Therefore, mechanisms for efficiently managing security are needed [25]. Security can be provided in different layers which will be introduced as follows:

- Network security:

  The network is considered one of the primary candidates for security threats. All sorts of information can be leaked if the network is compromised and therefore significant attention should be given to secure it. From a deployment perspective, there are certain ways in which security can be provided. One of the common methods is to introduce a firewall on the compute cluster. Access control list (ACL), secure access zones, port blocking, and blocking DoS attacks are few other options that can be used to make the security of a system better.

- Implement organization-wide security policies such as Multi-Factor Authentication:

  Authentication in clusters implemented when the users or services want to access the system through the API server. When a client request to the API server, the request gets checked by a list containing different plugins. These plugins try to obtain the identity of the client through different methods, such as client certificate, authentication token passed in an HTTP header, basic HTTP authentication, etc.

- Application data security:

  This can be achieved through password policies, access control mechanisms, encryption, etc. The level of security can be increased by securing the clusters through role-based access control which is introduced after facing a path traversal attack. This happens when clients try to retrieve the token to authenticate and run malicious pods on clusters [13].

4.2.4 *Performance:*

Several research has been done to improve the performance of applications using cluster computing. Through these research some models for performance and resource management have been suggested. In one of them, a Reference net-based model for Pod container lifecycle in Kubernetes has been suggested. [19] Such a model can be used as a basis to support: (i) capacity planning and resource management; (ii) application design, specifically how an application may be structured in terms of pods and containers. Moreover, Container Orchestration Platforms' teams are constantly improving the performance of the large-scale APIs of these systems and have derived tremendous improvements. When Kubernetes 1.2 was released, it supported clusters of up to 1,000 nodes within the Kubernetes service level objectives. Kubernetes 1.3 doubled the number to 2,000 nodes [34]. In order to achieve a better performance, we can summarize the findings to important points:

- Ability of system to provide adequate response time and throughput:

  In [4], a container performance study with Docker shows network performance degradation in some configurations and a negligible CPU performance impact in all configurations. Unlike Docker, Kubernetes uses a partial nested-container approach with the Pod concept where network virtualization is used once, as the same IP address is used for all containers inside a Pod, leading to better performance.

- Ability of system to handle peak loads during scalability:

  Although increasing the number of nodes in a cluster is key for horizontal scalability, pod density is important too. Pod density is the number of pods that can be managed efficiently on one node. If pod density is low, then running too many pods on one node does not help the scalability. That means that you might not benefit from more powerful nodes (more CPU and memory per node) because the node will not be able to manage more pods. In Kubernetes 1.2, the Pod density increased from 40 pods to 100 pods per node.

- Ability to define custom resources for containers:

  It is possible to define the amount of CPU and memory that a container needs (these are called Requests) and a hard limit on what it may consume (known as Limits). By defining the Requests, the scheduler will understand which Pod should be scheduled. If the requested amount of CPU/Memory was not available, the scheduler will not choose that specific Pod or any other Pod, which does not provide the Requested CPU/Memory, the workload, therefore, may be delayed to get done. In order to overcome this problem, QoS will be defined for each Pod's Requests/Limits. There are three different classes of Quality of Service; BestEfforts (lowest priority), Burstable, Guaranteed (highest priority). BestEffort means no guarantee for that pod to be provided by the desired amount of CPU/Memory and also for the sake of Limits, they are the first to be killed by schedulers. On the other hand, Guaranteed is the one with the highest priority in terms of providing Request/Limit features. Burstable pods get the number of resources they request but are allowed to use additional resources (up to the limit) if needed [18].

4.2.5 *Serviceability (Maintainability)*

Building an infrastructure is without a doubt a complex art that has been evolving over time. However, how to maintain such a system is another story. There are many aspects around that should be considered. Here we have summarized the two most important ones:

- Automation for the infrastructure creation using Infrastructure as Code (IaC):

  Maintaining infrastructure manually has many problems such as increased cost, inconsistency, speed, and many more. This once-considered manual process can unsurprisingly be automated in cloud computing. There are generally two major types, imperative and declarative. Where imperative instructions are given which should be executed at certain milestones while in declarative the outcome is told.

- Monitor Your Deployments:

For workloads to be production-ready, we need to have them monitored. Most production-ready charts include support for metrics exporters, so the application status can be observed by tools. Also, it is important to ensure that your workloads also integrate with logging stacks like ELK for improving the observability of your containerized applications. The advantages are uncountable: early failure prevention, auditing, trend detection, performance analysis or debugging, among others. [32]

4.3 Limitations

Despite doing all the configurations which a cluster can offer correctly and diligently, one can never rule out the importance of other factors such as coding ethics or the network bandwidth when talking about system quality. For example no matter how secure the deployed cluster is but if the underlying service did not follow coding conventions for protecting data, then data breaches and/or weak linkages can occur. One more example can be a data transfer between two services in which the underlying network plays an important role and deployment would not be able to mitigate the delays much. As seen from these examples it should be clear that no matter how good the deployment architecture is, there are several other factors that can always hinder the overall performance of the system and they should be given as much attention as the deployment architecture itself is given.

5. CONCLUSION

Compute clusters have positioned themselves as one of the fundamental concepts in the age of modern software development. These compute clusters help to achieve the desired benchmark of quality software must deliver such as availability time or scalability. This paper talks about how these clusters can be configured for deployment to deliver the quality of service (QoS) requirements that are expected from a certain software system. A relationship has been presented between different QoS requirements and clusters configurations along with various trade-offs between QoS requirements. The aim is to improve the quality of deployment by checking all the configurations one must keep in mind to achieve certain QoS requirements. The relationship presented in this paper helps to achieve this task by demonstrating all the configurations for a compute cluster along with certain external factors on which a certain QoS requirement might depend. This model makes it harder for the reader to miss out on any metric while ensuring a successful deployment as per need. Nevertheless, it includes some limitations as the deployment does not depend only on compute clusters but also on external features such as bandwidth. These external circumstances have not been discussed in this paper in detail to limit the scope. The model presented can be improved by future work, which considers these external factors and incorporates them in the model presented. An example for these features can be bandwidth or latency, upon the addition of which the model would be more informative and complete.

6. REFERENCES

[1] I. 25000. Iso/iec 25010.

[2] T. 500. Top500 news:, jun 2010.

[3] T. 500. Top500 news: Japan captures top500 crown with arm-powered supercomputer, jun 2020.

[4] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder. Performance evaluation of microservices architectures using containers. *2015 IEEE 14th International Symposium on Network Computing and Applications*, pages 27–34, 2015.

[5] F. C. Casaló, L.V. and M. Guinalíu. The role of security, privacy, usability and reputation in the development of online banking. *Emerald Group Publishing Limited*, 31(5):583–603, 2007.

[6] K. Casey. What's the difference between a pod, a cluster, and a container?, sep 2020.

[7] T. Chumash. Obtaining five "nines" of availability for internet services. *Rutgers University*, 2019, 2005.

[8] L. P. Dewi, A. Noertjahyana, H. N. Palit, and K. Yedutun. Server scalability using kubernetes. In *2019 4th Technology Innovation Management and Engineering Science International Conference (TIMES-iCON)*, pages 1–4, 2019.

[9] Docker. Build, ship, and run any app, anywhere, jan 2019.

[10] S. A. Ghobaei-Arani, M. An efficient resource provisioning approach for analyzing cloud workloads: a metaheuristic-based clustering approach. *Supercomput 77*, 2021.

[11] W. Hasselbring and R. Reussner. Toward trustworthy software systems. *Computer*, 39(4):91–92, 2006.

[12] I. M. A. Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, and A. Palopoli. Container orchestration engines: A thorough functional and performance comparison. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, pages 1–6, 2019.

[13] X. Jing and Z. Jian-jun. A brief survey on the security model of cloud computing. In *2010 Ninth International Symposium on Distributed Computing and Applications to Business, Engineering and Science*, pages 475–478, 2010.

[14] A. Khan. Key characteristics of a container orchestration platform to enable a modern application. *IEEE Cloud Computing*, 4(5):42–48, 2017.

[15] D. Kim, H. Muhammad, E. Kim, S. Helal, and C. Lee. Tosca-based and federation-aware cloud orchestration for kubernetes container platform. *Applied Sciences*, 9(1), 2019.

[16] T. Krazit. How zoom pulled off the scaling event of a lifetime, 2020.

[17] Kubernetes. Kubernetes documentation.

[18] M. Lukša. *Kubernetes in Action.* Manning, Pages: 445-450, 2018.

[19] V. Medel, O. Rana, J. a. Bañares, and U. Arronategui. Modelling performance amp; resource management in kubernetes. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, UCC '16, page 257–262, New York, NY, USA, 2016. Association for Computing Machinery.

[20] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das. Towards characterizing cloud backend workloads: Insights from google compute clusters. *SIGMETRICS Perform. Eval. Rev.*, 37(4):34–41, Mar. 2010.

[21] V. Nelson. Fault-tolerant computing: fundamental concepts. *Computer*, 23(7):19–25, 1990.

[22] A. Nordhoff. What is a cluster? an overview of clustering in the cloud, jul 2020.

[23] M. Palmer. Kubernetes networking guide for beginners, sep.

[24] S. Peng, H. Wang, and Q. Yu. Multi-clusters adaptive brain storm optimization algorithm for qos-aware service composition. *IEEE Access*, 8:48822–48835, 2020.

[25] M. Pourzandi, D. Gordon, W. Yurcik, and G. Koenig. Clusters and security: distributed security for distributed systems. pages 96– 104 Vol. 1, 06 2005.

[26] S. Prathiba and S. Sowvarnica. Survey of failures and fault tolerance in cloud. In *2017 2nd International Conference on Computing and Communications Technologies (ICCCT)*, pages 169–172, 2017.

[27] R. A. P. Rajan. Serverless architecture - a revolution in cloud computing. *Tenth International Conference on Advanced Computing (ICoAC)*, pages 88–93, 2018. 10.1109/ICoAC44903.2018.8939081.

[28] Redhat. What is a kubernetes deployment?

[29] Redswitsches.com. The difference between horizontal vs vertical scaling, dec 2019.

[30] D. Rountree and I. Castrillo. Chapter 6 - evaluating cloud security: An information security framework. In D. Rountree and I. Castrillo, editors, *The Basics of Cloud Computing*, pages 101–121. Syngress, Boston, 2014.

[31] N. Sadashiv and S. M. D. Kumar. Cluster, grid and cloud computing: A detailed comparison. In *2011 6th International Conference on Computer Science Education (ICCSE)*, pages 477–482, 2011.

[32] J. Salmeron. 5 tips to deploy production-ready applications in kubernetes.

[33] G. Sayfan. *Mastering Kubernetes.* Packt, Page: 215, 2017.

[34] G. Sayfan. *Mastering Kubernetes.* Packt, Page: 238, 2017.

[35] W. Staff. Cloud containers, may 2021.

[36] J. E. Victor Millnert. Holoscale: horizontal and vertical scaling of cloud resources. *IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, page 196, 2020.

[37] X. Wang and H. Wang. Driving behavior clustering for hazardous material transportation based on genetic fuzzy c-means algorithm. *IEEE Access*, 8:11289–11296, 2020.

[38] Wikipedia. Cluster computing.

[39] Q. Wu, J. Yu, L. Lu, S. Qian, and G. Xue. Dynamically adjusting scale of a kubernetes cluster under qos guarantee. In *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 193–200, 2019.

Towards a Quality Model for Messaging Systems in Event-Driven Architectures

Noyan Ahmed Siddiqui
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
noyan.siddiqui@rwth-aachen.de

Laurens Studtmann
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
laurens.studtmann@rwth-aachen.de

ABSTRACT

Event-driven architectures (EDAs) are a highly important paradigm, used in a lot of critical business applications in need of fast, scalable, reliable, and asynchronous communication. This includes the likes of stock exchanges, news tickers, air-traffic control, and supply chain management. These and similar applications need to be able to react quickly to their often very dynamic environments, and can thus make use of the advantages of event-driven architectures. A key component in any event-driven architecture is the messaging system responsible for delivering event notifications. Because such a system has great influence on the behaviour and characteristics of the EDA, it is highly important to consider the quality of the messaging system.

In this paper, we set out to create a clear definition of quality for messaging systems in event-driven architectures by constructing a quality model. We offer a list of external quality attributes for messaging systems that may need to be catered to when applying an event-driven architecture to an application. The quality attributes making up our model, concerning characteristics like efficiency and reliability, are described and explained in detail. We also explain how the attributes can improve the quality of the application, how to apply them, where trade-offs need to be made among them, and give a rationale to which extent an attribute is fulfilled. We showcase how these attributes can be applied and how they interact with each other on the examples of Kafka and RabbitMQ, two widely used message brokers in the industry.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

Keywords

event-driven architectures, messaging systems, Kafka, RabbitMQ

1. INTRODUCTION

There is great demand for software that can handle asynchronous communication between different components based on significant changes in the environment of a system, also referred to as events. Event-driven architectures (EDAs) try to address this exact need. In an EDA, participating agents detect events and can react by sending a message, also called event notification, which can be received by other agents in the system. Many modern-day applications are in need of such systems, like for example stock exchanges, news tickers, air-traffic control, and supply chain management [3][15]. There are many factors that determine the quality of an EDA for a given application, and one major part is made up by the messaging system chosen for delivering the event notifications. The quality of that messaging system, described by characteristics like end-to-end messaging latency or fault tolerance, is thus highly important for the overall quality of the application.

1.1 Research Goal

Creating a detailed quality model that describes and explains quality attributes in messaging systems for EDAs is an important contribution to the development of better EDAs. As such, it is the goal of this paper to present such a quality model for messaging systems in event-driven architectures. For that we have set the following four objectives: First, to explain the quality attributes themselves, second, to answer the question of when and why they are important, and third, where there are trade-offs to be found between the different characteristics. To further illustrate this, we have chosen two popular messaging systems, Kafka and RabbitMQ, for our fourth objective: To explain for the different characteristics how they can be catered to in the real world, considering the relationships among them in particular.

For this paper, we focus only on external quality attributes. Internal attributes like maintainability or interoperability are out of scope.

The paper will first introduce and explain the general functionality and structure of event-driven architectures, give an overview of both Kafka and RabbitMQ, and then go over quality models in general. The main part will focus on the quality model, covering all the different attributes in detail as well as how they can be handled in Kafka and RabbitMQ. Section 4 will conclude the paper and offer some avenues for further research.

2. BACKGROUND

2.1 Event-Driven Architectures

An event-driven architecture (EDA) is a software design pattern in which participating agents are categorised into producers and consumers. Producers react to events, which are significant and relevant changes in the environment of the software system, by packaging the relevant information into a message, also referred to as an event notification, and sending it over a messaging system. An event notification is handled by a so-called broker and transmitted to consumers which can then act upon it. A notable property of most EDAs is that producers and consumers are only loosely coupled. This enables event-driven applications to become highly scalable, an important property in particular for large distributed systems.

The communication model of EDAs is considered message-oriented. This means that it is based around asynchronous messaging instead of a request/response model. Such asynchronous communication is necessary because events can occur arbitrarily and need to be distributed and dealt with at any time. The message-oriented communication of EDAs can generally be categorised into the point-to-point (P2P) model or the publish/subscribe (pub/sub) model [14]. With P2P, each message sent by a producer goes through a queue which relays the message to one specific consumer. When EDAs follow the publish/subscribe paradigm, the messages are assigned to categories by the producer (now called publisher). A consumer (now called subscriber) can subscribe to a category that is relevant to them. When a publisher sends a message of that category, all the agents subscribed to that category will receive it. Compared to P2P, this further anonymises the agents and increases their independence: Publishers only assign categories, not specific agents, and subscribers only subscribe to categories and have no knowledge of the publishers who sent the messages. This strong decoupling further improves the scalability of EDAs.

2.2 RabbitMQ

RabbitMQ [4] is a popular open source messaging system implementing the Advanced Message Queuing Protocol (AMQP). AMQP was originally developed for the finance industry as a standard for asynchronous messaging, and was thus shaped by the strict performance, reliability, and scalability requirements of that sector [5]. RabbitMQ builds upon this by extending AMQP in a few different ways: For example, RabbitMQ can let clients know that a connection is blocked or unblocked, it can prioritise certain consumers so that they are notified first, and the system can ensure that rejected or expired messages are rerouted (so-called dead lettering). For a more exhaustive list, see the protocol extensions in [11]. Nowadays, RabbitMQ supports protocols other than AMQP via plugins, like the Simple Text Oriented Message Protocol (STOMP) for example. RabbitMQ is written in the Erlang programming language, which is designed for building distributed and fault-tolerant software [4]. Due to the use of the actor model in the communication between different Erlang processes in the underlying system, scalability is enhanced significantly [5].

These general design decisions are supposed to enhance the quality of applications that use RabbitMQ as the message broker in event-driven architectures. How exactly different quality characteristics are affected will be discussed in each respective part in Section 3.

RabbitMQ's messaging system revolves around so-called exchanges, which act like mailboxes or post offices in the real world. An illustration of the components and their interaction can be seen in Figure 1 (b). When a publisher sends a message, it is delivered to an exchange, which handles the distribution of the message to queues according to certain rules (referred to as bindings). The content of the queues can then either be fetched by consumers or the messages are directly pushed to consumers subscribed to the queue in question. Depending on the type of exchange, both point-to-point and publish/subscribe models for delivering notifications can be realised. The exchanges and queues are stored on and handled by RabbitMQ nodes, which are somewhat independent instances of RabbitMQ, that make up the messaging system.

2.3 Kafka

Apache Kafka is a distributed event streaming platform capable of handling large amounts of data in the form of events. It is a highly popular system, handling trillions of events per day [2]. It started off as a messaging queue service but quickly evolved to a full-fledged event streaming platform. It provides, among others, the following core functionalities [2]:

- Publish and subscribe to a stream of records
- Store the streams of records, in categories called topics, in a fault-tolerant way
- Process streams of records as they occur

The earlier discussed concept of producers and consumers applies to Kafka as well. Producers can publish a stream of records to one or more Kafka topics and on the other end, consumers can subscribe to topics and consume streams of records produced for them. A topic is the core abstraction that Kafka provides for a stream of records to categorise the records that can be published to the topic. A topic itself can have multiple subscribers at a time. A topic is further partitioned, i.e. it is spread over a sequence of buckets of records located on different Kafka servers, also known as brokers. A certain record stored within a certain partition of the topic is assigned a sequential ID, called offset, that identifies that record.

Kafka is used for following broad class of applications:

- Building real-time streaming data pipelines that reliably get data between systems or applications
- Building real-time streaming applications that transform or react to the streams of data

2.4 Quality Models

Quality models provide a framework that evaluates a system's overall quality and value it provides to the end users. It aims to describe the quality of a system, meaning the extent to which the system is valuable, meeting the goals and objectives of the application. The idea is to first define the goals of the product. The identification of the goals/objectives helps to set priorities based on the importance of each quality aspect.

The goals can then further be mapped to concrete quality attributes, also called quality characteristics, which help with evaluating the quality of the system. Each of these characteristics can also be decomposed into subcharacteristics if they are too broad. Furthermore, for each of the

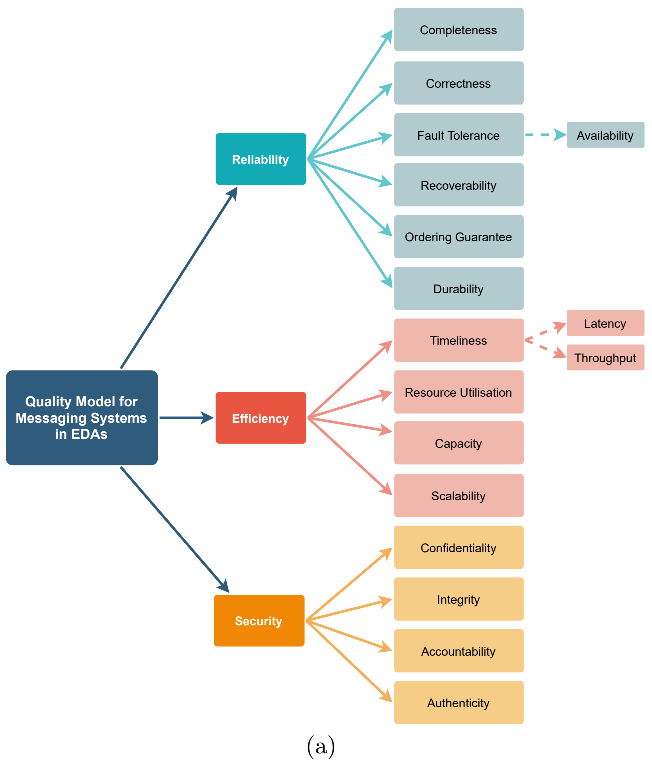


Figure 1: (a) Diagram visualising the attributes of our quality model, organised into their different categories. (b) Illustration of basic components in the RabbitMQ messaging system, adapted from [4].

quality attribute/sub characteristics certain metrics can be identified along with some measurement criteria. However, determining quality metrics is out of scope for this paper.

2.5 Related Work

Papers that deal with designing and developing modern event-driven applications in general without our chosen focus on the underlying messaging system include [14] and [10]. [1] and [3] also deal with EDAs and publish/subscribe middleware respectively from a Quality of Service perspective, and offer general insights into improving these types of systems as a whole. These works differ from our paper in the sense that we focus on messaging systems and put emphasis on the relationship between the different quality attributes.

For a comparison between different messaging systems and deciding which broker to pick in which kinds of applications, there is [7] which compares Kafka, RabbitMQ, RocketMQ, ActiveMQ, and Pulsar. Other works featuring similar comparisons are [5] between Kafka and RabbitMQ, and [8] between RabbitMQ and ActiveMQ. These papers cover the mentioned systems in a general, non-EDA specific context. Also note that due to the age of the papers and the rapid ongoing development of the messaging systems, some newer features are not considered in these works which we do include in our findings, like for example quorum queues in RabbitMQ. For more of a general concept and motivation behind RabbitMQ, see [4].

3. QUALITY ATTRIBUTES

In the following section we present the quality model for messaging systems in EDAs. A visualisation of the quality attributes in their respective categories can be found in Figure 1 (a).

3.1 Reliability

One of the most important requirements and signs of quality in a software architecture in any business application is reliability. It is key that the software does what it was designed to do consistently and with as low of a probability of failure as possible, even during adverse circumstances.

With EDAs this means that the handling of events and event notifications by producers and consumers and the transmission of messages between them happen in a reliable fashion. This, however, is a challenging task in the dynamic environments where the event-driven approach is most suitable: Events occurring at arbitrary times with large changes in the number of notifications sent at a given moment, varying message sizes, changes in the list of consumers that need to receive the notifications, as well as the need for asynchronous transmission of messages often across multiple network hops are all sources of non-determinism in the system. They are common challenges in the real world, especially when paired with limited processing resources [10]. When reliable transmission of information is critical, as is the case in a lot of business applications, dealing with these concerns becomes a high priority. Ensuring the quality of a software based on an event-driven architecture means catering to the following reliability criteria concerning the messaging system.

3.1.1 Completeness

Perhaps the most obvious quality attribute related to reliability is completeness, also referred to as at-least-once delivery, meaning that all event notifications are received and dealt with by the respective consumers, with no messages lost along the way.

Lost messages can have strong negative effects on the application, like when critical information on sudden changes in the environment stays unknown to relevant parts of the system.

A loss of messages can be caused by inconsistencies and non-determinism in the network, network failure, or errors in the producer, consumer, or the notification delivery software in between [3].

Factors that can provoke or amplify such problems include an overload of the system – e.g. with a large number of events and/or large message sizes –, unreliable network infrastructure, or events that are not properly handled by the software – e.g. incorrect event types or corrupted messages. It is the responsibility of the messaging system to account for such occurrences and handle them gracefully. For further detail on handling of larger failures, see Section 3.1.3 on fault tolerance.

As a quality metric, instead of the sometimes infeasible all-or-nothing requirement that is completeness, one should rather consider the probability of delivery of the event notifications [3]. Of all the notifications that are supposed to be delivered to a consumer, what is the fraction of the messages that arrive?

To increase the probability of delivery one can make some immediate considerations: The protocol for the underlying communication can be chosen to be reliable. Reliable protocols like for example TCP [13] can support the detection of lost, incomplete, or corrupted messages, and resend them or apply error-correction techniques to ensure a proper delivery of the message. But such protocols usually come at the expense of some efficiency. These features generally require more information to be sent, and more processing to take place for sending, transmitting, and receiving a message. In an environment with limited resources, this can cause not only performance issues but in extreme cases even an overload or outage of the system and thus a decrease in reliability if the measures taken for ensuring completeness are excessive.

It is a highly application-specific balance that has to be struck between acceptable loss of messages and efficiency.

The protocols supported by RabbitMQ all use TCP for communication. This means that on the network level, messages will be guaranteed to be transmitted, as long as the TCP connection is upheld. However, any operation above the network layer must still be considered for completeness, which is why one can choose to use consumer acknowledgements and publisher confirms in RabbitMQ to ensure that the messages are not only received, but also acted upon by both the broker and the consumer. Effectively, these acknowledgements and confirms show a change in ownership of the particular message that was acknowledged or confirmed. In the end, this guarantees every message to be delivered at least once, and means that completeness is fulfilled, as long as acknowledgements and confirms are used. But, as discussed in the next section on correctness, fulfilling the at-least-once delivery guarantee means losing the at-most-once delivery guarantee in RabbitMQ [11].

In Kafka [2], completeness in the sense of at-least-once-delivery is handled similarly, by resending the message until confirmation of the arrival is received. This also covers the case of the transmission of a large file which fails half way through the processing. It will then resend the message and process the file again from scratch. In any case completeness with at least-once-delivery is ensured. However, after the successful processing of a message, it can happen that the acknowledgement is lost, causing the message to be sent twice, violating correctness.

3.1.2 Correctness

Correctness can be considered a counterpart to completeness. A system is correct if all received messages are messages that were supposed to be received.

Catering to correctness means to avoid sending unnecessary messages, like event notifications without the associated event occurring due to an error in the producer's software. Such false-positive messages are comparatively hard to come by though, as arbitrarily sent messages without an actual event occurring would be rare in a functional event-driven application. In the real world, however, duplicate messages will be the most common type of superfluous message to deal with, as they can occur randomly due to network-related errors and attempts at resending lost messages for example [11]. This would then violate the notion of at-most-once delivery, where these duplicates are not allowed.

Depending on the application and the type of event notification that was duplicated or otherwise superfluous, problems with correctness may not be as impactful as a missing message. For example, when an event notification informs about the current ambient temperature, a duplicate message has no serious consequences other than the use of slightly more resources. In other cases, it could mean that an article is shipped twice instead of once or that some information is displayed redundantly in a user interface.

That does not mean however that unnecessary messages cannot cause serious problems: If too much of a medication is sent to patients at a hospital due to duplicate event notifications, patients may take too high of a dose, or the supply of the hospital would run out unexpectedly, causing some patients to not get their appropriate medication.

So it can still be important to cater to correctness, by reducing the amount of superfluous messages or avoiding them entirely. As with completeness, the protocol of the underlying communication channel can be set up to detect and deal with such messages at the cost of some compute resources and bandwidth [13]. Or one can use the strategy of "fire-and-forget" to guarantee at-most-once delivery because messages are always transmitted just once [11]. But this would considerably decrease the probability of delivery, as a lot of measures for ensuring completeness like resending notifications would need to be excluded from the system.

Having a system that is both complete and correct would guarantee exactly-once delivery, meaning that exactly one message is sent and received for every event notification that is supposed to be transmitted. While this is the ideal scenario, it is challenging to achieve in all circumstances.

For example in RabbitMQ, at-most-once delivery can only be achieved by choosing not to use acknowledgements and confirms, and delivering messages in a "fire-and-forget" fashion. But because the acknowledgements and confirms are required for at-least-once delivery, guaranteeing exactly-once delivery is not possible in RabbitMQ [11].

Similar to RabbitMQ, the at-most-once delivery type can be achieved in Kafka by following a "fire-and-forget" policy. In addition to that, Apache Kafka also supports exactly-once stream processing through Streams API, by enabling the *exactly-once* processing guarantee in the configuration [2].

3.1.3 Fault Tolerance

Hardware, infrastructure, and software outside of the event-driven architecture can and will fail in some way for most systems eventually. In order to make an EDA properly reliable, its messaging system needs to be able to robustly handle these cases of failure as gracefully as possible.

Improving the fault tolerance of an EDA means identifying the most likely points of failure in and around the application, and making sure the messaging system keeps working to the extent that is possible in the given circumstances if such a failure occurs. In most applications, a system that crashes entirely when such issues emerge is worse than a system that drops a few messages, which is worse than a system that becomes temporarily slower to deliver notifications. In case of a network failure or crash of a participant, for example, the system should not crash or ignore the issue, but detect the problem and engage in measures for recovery, as explained in Section 3.1.4 on recoverability. In case a producer crashes, the system could detect this and notify relevant consumers of the problem. Or if a consumer crashes, durable messaging as discussed in Section 3.1.6 can still notify the consumer of the missed messages after it comes back online.

A more extreme measure that also requires extra hardware is the support for redundant communication channels. By transmitting over multiple connections, the system can still operate even if one channel fails entirely. This would also have the consequences of much increased resource utilisation as well as not being able to guarantee at-most-once delivery, due to every message being transmitted multiple times.

Multiple instances of a RabbitMQ server within the same local network can be configured to function as a single broker, which enables better fault tolerance, as well as load balancing across instances at the cost of performance and resources [4].

A queue mirroring technique called quorum queues can be used in RabbitMQ to improve the fault tolerance when data safety is of high importance [11]. Using quorum queues means that a number of replicas are created for each queue, with one designated leader. In case the RabbitMQ node of the leader fails or becomes otherwise unavailable, a new leader will be elected which takes over the responsibility of the old leader. This way, the system can withstand the failure or unavailability of multiple nodes, as long as there are enough nodes left that are responsible for the particular queue to form a majority. Configuring a queue to use this feature requires more resources and limits scalability in particular, as keeping the replica queues up-to-date is a comparatively expensive and time-consuming operation.

Kafka streams are built on fault-tolerance capabilities. A multi-datacentre approach for Kafka deployments allows the system to disperse to data-centres in multiple regions [2]. In case of a disaster – power outage, software failure, hardware failure, or any other error causing a datacentre to fail completely – Kafka continues to work on other active datacentres until the service is restored.

Improving the **availability** of an event-driven application means making sure that the system is operational and responsive as often as possible. In other words, system downtime needs to be reduced.

Availability is an important consideration in an event-driven architecture and as such its messaging system. When the brokers go down, the whole system may become unresponsive. So for a lot of applications, the availability of the messaging system is very high on the list of priorities.

Catering to availability means increasing the fault tolerance of the system, making availability a consequence of fault tolerance, so the techniques described for fault tolerance 3.1.3 apply. Fault tolerance encompasses more than improving availability, though, as availability is about the time the system is functional at all, while improving fault tolerance can also mean lessening the performance impact of an occurring fault, which has no impact on availability.

3.1.4 Recoverability

In case of some parts of the system failing catastrophically, a messaging system can be designed to be able to recover lost information and continue operating after the issue has been resolved.

This may be important for applications where critical information is being transmitted and data loss would have negative impacts in addition to the problems introduced by the system failure. With all the data that is saved in a system with high recoverability during operation right before a failure occurs, it is also a helpful property for figuring out what caused the issue in the first place.

For example, if parts of the system responsible for the transmission of messages fail and crash, any event notifications in transit may be lost, as they are generally only kept in a volatile memory buffer for a short period of time. A way of making the system more recoverable would be to use persistent messages [15]: Instead of only keeping the messages in memory, they are also written to a more permanent storage solution, like a database. This way, they can be retrieved and retransmitted after an outage.

This comes, like most other improvements to reliability, at the cost of more processing resources and latency. Additional hardware components are also required for the persistent storage.

In RabbitMQ, individual messages can be declared to be persistent, in order for these messages to be written to disk by the so-called persistence layer [11]. To gain the benefits just described, the queue that the message was placed in also needs to be configured to be durable, so the queue stays and is recovered after the node in question is restarted. In the other queue mode, called transient, the messages in the queue are lost, even if they were declared persistent. Durable queues are not to be confused with the notion of durability we describe in Section 3.1.6, however.

Partitions in Kafka are highly available and replicated to multiple instances [2], as shown in a possible architecture for a multi-datacentre deployment of Kafka in Figure 2. In case of a failure or if re-processing is required, the stream data is persisted to Kafka. Kafka also maintains a replicated changelog for every Kafka topic which tracks all state updates. These changelogs topics are further partitioned as well so that each local state store instance. Next when the consumer tries to reprocess the failed process the data is still available.

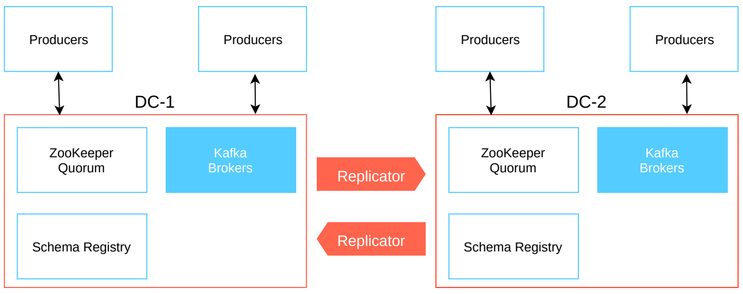


Figure 2: A multi-datacentre reference architecture [2].

3.1.5 *Ordering Guarantee*

Fulfilling the ordering guarantee means that messages are received in the same order they are published.

This order of messages may have a significant impact on the functionality of a system. If the order changes during transmission, one can imagine the following scenario:

A machine in the system controlled by the EDA is powered off. An event notification is sent to turn the machine on, and a split second later, another notification is sent to turn it off again. If the messages are received in reverse order, the consumer will presumably try to turn off the already powered down machine, which may or may not cause problems already, and then turn it on. The machine may now be running indefinitely even though it should not be, until the error is noticed or another shutdown event notification is received.

Similarly to completeness and correctness, the underlying communication protocols like TCP can do most of the work ensuring correct ordering of messages, at the cost of some overhead [13].

As TCP is in use in RabbitMQ, any ordering issues on the network layer are taken care of. Above this layer, ordering is preserved per default under the assumption that the messages are consumed by a single consumer only [11]. The reason for this limitation is that if there are multiple subscribers of a queue, one consumer may requeue a message, causing that message to be out of order. So for messages that require strict ordering, one needs to resort to a queue bound to only a single consumer. However under this assumption, the ordering is preserved even under the circumstances of the communication channel closing or the message being requeued by the consumer.

In Kafka, topics are split into partitions. The ordering of messages is preserved within the partition by default, but cannot be guaranteed across partitions. In case of multi-partitioned topics a system of key/partitions can be used to maintain the order of messages. The keys used by Kafka allow for messages with the same key to be put on one partition [6].

3.1.6 *Durability*

In the pub/sub paradigm, a messaging system is considered durable if it fulfils the following functionality: When a consumer is subscribed to a category of messages but becomes inactive – for example due to a restart of the consumer –, the messages they would have received during the downtime are stored. When the consumer in question comes back online, the missed messages will be delivered [15].

Other causes for this feature to become relevant would be a crash of a consumer or network failure. In both cases, the consumer would not be reachable, and the messages would be kept until they can be delivered. This comes at the cost of some memory and/or disk space, which can become a problem if a large number of messages are supposed to be delivered to inactive consumers, like when a consumer is shut down for a long period of time.

Do note that the concept of durability we refer to here is not the same as persistent messaging, as persistence is about the broker responsible for transmitting the data failing or being otherwise powered off, while durability is about the consumers being temporarily unavailable. Persistence is also mostly aimed at system failure or other rather unplanned shutdowns, while durability may be a more normal part of the workflow, with consumers powering down without unexpected causes, to save power for example, and still want to consume the missed messages the next time they are active [15].

In RabbitMQ, all messages are durable by default in the sense just described. However, durable queues in RabbitMQ refer to a different concept more related to persistence, and are not to be confused with the notion of durability explained in this section. It is further discussed in Section 3.1.4.

As mentioned earlier, by default the stream data in Kafka is stored in a persistent manner. The use of distributed commit logs/changelogs makes it durable.

3.2 Efficiency

The efficiency of a software system is the ability to fulfil all the functionalities of the system and its purpose with optimal utilisation of all the necessary resources including time, memory, storage, etc.

Efficiency can be considered inherent to the fundamental design of event-driven architectures. EDAs consist of modular participants, namely producers and consumers, that can work simultaneously without being dependent on each other. Given the modularity, even multiple consumers and producers can spin up and work in parallel. This loose coupling of an EDA's participants results in more seamless, scalable, and in the end efficient operations. A fully event-driven system is well-suited for asynchronous communication and therefore also for parallelisation [14].

To deliver an efficient solution in an EDA, a middle-ground has to be identified for the particular application between resource utilisation and time consumption.

3.2.1 *Timeliness*

Time behaviour in event-driven architectures refers to the latency and throughput rates that the system supports and guarantees to perform activities. Ideally, an efficient EDA enables high throughput while offering low latency.

The **latency** of an event being handled can be a measure of multiple different time spans in the operation of an EDA: The latency we are interested in is the raw delivery time, or the time it takes from the producer sending the event notification to the consumer receiving it. This concerns only the messaging system used for communication, without considering the processing time within the producers or consumers, which of course also has an impact on time behaviour. For this paper, however, we are only concerned with the messaging system.

Efficient time behaviour is obviously a highly important aspect of a messaging system, as a lot of applications have the need of events being dealt with quickly. The lower the latency, the faster the system can adapt to changes in the environment, increasing the quality of the system considerably. Some systems have very strict real-time requirements that need to be catered to, where an event needs to be dealt with within a set amount of time. This means that not only is there a requirement for the latency being low, but also consistent [10].

There can be multiple ways to reduce and stabilise latency. The most obvious area to configure is the chosen communication protocol. Using faster protocols can improve the latency, however often at the cost of reliability and/or resource utilisation. Using UDP [12] instead of TCP [13] for example will positively impact latency, but cause a decrease in reliability.

Apart from that, efficient event filtering can also positively impact the latency [10]. If events are categorised more finely, they can be sent only to the consumers that need it, saving time and resources by reducing unnecessary transmissions. On the flipside, having the categories be too fine – and thus too numerous –, encoding and matching the events into the categories can become processing intensive. Defining a proper category to an event can also become more error-prone, especially if it is done by a user. This means that a proper balance needs to be found between coarse and fine-grained categories. A highly sophisticated approach would be to adapt the granularity of categories depending on the current conditions [10]. If event filtering is the bottleneck because of limited processing capabilities in some producers, the model should become more coarse, but if the network transmission is the limiting factor, finer-grained categories should be used.

With RabbitMQ, one has a lot of freedom to define categories as sophisticated or basic as it is required, due to the use of the routing system of AMQP. Flexibility is further increased by the different kinds of exchanges that can be used to fulfil not only the basic point-to-point or publish/subscribe needs [11].

The routing in Kafka is much more restricted, which can lead to suboptimal distribution of messages and thus a performance decrease in some applications in need of more sophisticated routing [5].

Throughput refers to the amount of data being transferred per unit of time. For larger messages, this may be the factor to prioritise when it comes to timeliness, while latency is more important for smaller messages. However, the two concepts are of course heavily intertwined.

A way of improving throughput in event-driven architectures can be applied if the job contains multiple independent sub-tasks. These subtasks can then be taken over by different consumers and thus result in better throughput, and thus faster completion of the whole job with reduced time consumption. Adding more consumers further emphasises this improvement, however the overhead introduced by having to synchronise over the larger number of consumers may degrade performance significantly [15].

As fundamentally distributed messaging systems, both RabbitMQ and Kafka benefit in terms of throughput by the use of distributing tasks across multiple machines.

3.2.2 Resource Utilisation

This quality attribute is concerned with using the least amount of hardware, processing power, memory, and storage. The asynchronous nature of event-driven architectures, can allow for better utilisation of resources needed to perform a job.

A scenario in this regard can be found for example in synchronous systems having multiple microservices: When a process needs to be executed or some data needs to be fetched from a service then a call is made to that service. The call can take time to respond depending on the complexity of the process or the size of the data to be fetched. While the service is processing the request, the client (caller service) is in a blocked state waiting to get the desired response. During this time the client cannot accept or process any other task. Only when the server (processing service) gives back the response and the client receives it, can it move on to another task. This results in a poor utilisation of resources.

On the contrary in EDAs, this case is handled using state-driven asynchronous communication. The client in the EDA is the producer that produces and sends an event notification. Once the message has been sent, the job of the producer is done. It can now move on to work on other pending or new tasks. The server, which acts as a consumer, during this time will consume the task from the event notification delivery system and start processing it. Once the consumer is done processing the task, it now has two options. First, if the client needs to get notified of the process termination status then the consumer will now become a producer and push a message for the client informing about the status of the task, whether it was completed, if there were any errors, or similar. Second, if the producer does not need to know the status of the task then the consumer will move on to another task. This way both the client and the server do not have to wait on each other while any process is being executed and can work in parallel [15].

Making proper use of this asynchronous delivery of messages to more efficiently use the available resources is a significant advantage for EDAs.

A problem when it comes to resource utilisation can occur if there is an event that happens very frequently, to the point of becoming almost a continuous change in the environment. For example, a sensor can measure the temperature at a rate of perhaps ten times per second. Because the temperature always changes slightly, each of these measurements can be considered an event, for which an event notification would be produced. Depending on the application and available resources, this high frequency of very similar events may be unnecessary and possibly too straining on the system, causing higher latency or perhaps even an overload of the system. A way of reducing this would be a feature of the producer to set a minimum sampling rate [3]. So if the producer notices an event, like a change in temperature, but a very similar event has been processed and sent as a notification less than a set time ago, it will not produce another notification, conserving resources.

The resource utilisation is also impacted by the way state management is handled in EDAs. Redundant storage of the same data distributed across different modules has a negative impact on resource utilisation as it requires more resources to store the same data.

For RabbitMQ, resource utilisation can be reduced by turning off various features like e.g. persistent messages and durable queues (at the cost of recoverability), or consumer acknowledgements and publisher confirms (at the cost of completeness), or mirrored queues (at the cost of fault tolerance), among others. One can also reduce the number of RabbitMQ nodes in the system (at the cost of reliability, timeliness, or both) [11].

Due to Kafka's high scalability, as explained in Section 3.2.4, resource utilisation can be handled very well. Depending on the requirements, multiple consumer or producer nodes can be configured to spin up or down. The loose coupling of modules allows us to use resources efficiently as needed. Resources can also be more efficiently utilised by taking traffic in account, i.e. adding or removing resources to or from the cluster based on events and data traffic. Configuring replication and partitioning levels also assists in efficient resource utilisation [2].

3.2.3 Capacity

Capacity refers to the limits of a system in terms of how much activity it can handle [9]. It is determined by how resource efficient the system is and how many resources are available.

In event-driven architectures, capacity can be thought of as the number of events, producers, consumers, subscriptions, unsubscriptions, as well as event notification sizes that the system can handle at a time. Is the designed architecture powerful enough to accommodate all of these factors simultaneously? Is the event notification delivery system robust enough to handle the required number of events? These questions will help answer whether the capacity of the designed system based on event-driven architecture actually meets the requirements or not. It is therefore very important to keep capacity in mind while designing an event-driven system otherwise the advantages of EDA can turn out to be disadvantages. Event queue overflowing can be taken as an example in case more than the acceptable number of events are fired at the same time.

If capacity is reached, it may mean catastrophic failure, temporary outages or slower service, depending on how fault tolerant the system is, as discussed in Section 3.1.3.

RabbitMQ increases its capacity in memory-constrained scenarios per default by using the persistence layer [11], which is employed for making messages more recoverable in case of failure by writing them to disk, as discussed in Section 3.1.4 on persistence. This layer can also be asked to write non-persistent (so-called transient) messages to disk, in order to save memory. This of course comes at the cost of latency and throughput, as writing to and reading from disk is much slower compared to performing the same operations to and from memory. It also causes more I/O operations. As a consequence, this feature only activates at times when the memory is not large enough to hold all the messages.

The concept of lazy queues in RabbitMQ is another way of improving the capacity of the system in the following specific scenario [11]: They are to be used in the event of large amounts of messages accumulating in the queues, due to downtime of consumers, a sudden increase in messages, or slow consumers. They can accommodate these queue sizes by writing the messages to persistent storage as soon as possible instead of waiting for the memory to become full. The increase in I/O operations and the inherent slowdown caused by this are not as important in the intended use-case, as message consumption is comparatively slow anyways. But queues are not configured as lazy per default for these performance reasons.

In Kafka, there are multiple ways to handle capacity depending on the needs. If capacity is taken in the sense of how much or for how long logs and data can be held by a Kafka cluster, then this can be catered to by configuring the topic's retention period, log compaction and compression strategy, and also the replication factor. If the concern is how much data the Kafka cluster can handle, then configuring AWS instance types or the number of nodes can solve this issue [2].

3.2.4 Scalability

Scalability is a major reason why event-driven architectures are chosen in the first place. When the application demands the ability to add or remove resources and participants dynamically, as is often the case in distributed systems, scalability is the attribute to look for in a software pattern.

This inherent advantage of EDAs stems mostly from the fact that the participants of the system, the producers and consumers, are only loosely coupled. This independence of components means that adding additional producers and consumers is comparatively simple, with little impact on the rest of the system [15].

One of the predominant factors that determine the scalability of an EDA is whether the system uses a point-to-point or publish/subscribe model for messaging. Because the publish/subscribe model decouples the participants heavily, using it improves scalability considerably. But point-to-point models enable stricter enforcement of latency constraints within the system, due to its more straight-forward passing of messages from one producer to exactly one consumer. This means that there is a trade-off between scalability and the ability to fulfil latency constraints [3].

This trade-off is apparent when considering the quorum queues in RabbitMQ: Because of the strict replication of the queues' contents when this mode is activated, as described in Section 3.1.3, scalability is inhibited in order to increase reliability. As this setting is applied on a per-queue basis, one can use a small number of highly reliable quorum queues for messages where reliability is key, but resort to normal queues everywhere else so as not to limit the resource consumption and scalability of the system too much [11]. The appropriate balance is of course determined by the needs of the specific application.

Scalability is also an inherent feature of RabbitMQ's implementation due to Erlang's actor model for concurrent computation [5].

Kafka can handle scalability in all the four dimensions, i.e. event producers, event processors, event consumers, and event connectors. In other words, Kafka scales easily without downtime. Scalability of data consumption can be taken as an example. Adding new consumers helps in parallelising the data processing. Also by default, the distributed placement of records over different partitions within a topic enhances scalability as well as allowing multiple client applications, both consumers and producers, to read and write data from multiple brokers simultaneously [2].

3.3 Security

Security is a highly important aspect of any software system. A traditional monolithic system is a solid and stable element of infrastructure. A small change or error or a malicious attack on the system can crash the whole system in one strike. Despite having this huge disadvantage, the errors or the cause of the crash are easily traceable.

On the contrary in EDAs, where everything is divided into independent blocks and modules, a security fault in one of the components will mainly affect that component. But the error traceability is somewhat of a challenge. Since EDAs revolve mainly around network communication, applying aspects of network security can give us a solid foundation for providing security for the whole system [11].

3.3.1 Confidentiality

Confidentiality is the property of a system to protect its data and services from unauthorised access. Data shall only be accessible to those authorised [9].

In the EDA world, this means that all the data, services, and components should only be accessible by legitimate authorised modules. Every producer has to go through an authorisation step before being able to read any data like event categories from the system. Similarly and perhaps more importantly, the consumer also has to be authorised to be able to consume an event from the notification delivery system to ensure confidentiality.

Improvements to confidentiality can be made by introducing some form of encryption, which may come at the cost of efficiency, as more resources are needed for encrypting and decrypting the information. In a proper end-to-end encrypted scenario, even if the transmitted messages are intercepted, the unauthorised entity would not be able to tell the content of the message.

Confidentiality is a very important quality attribute in the current data-centric world, as the handled data may be private, or otherwise critical in nature, and should only be seen by explicitly authorised participants.

Encryption of data via the TLS encryption scheme can be turned on in RabbitMQ to ensure confidentiality [11].

RabbitMQ also employs a permission system as standard that authorises what actions a client is allowed or not allowed to perform. Firstly, the resources of a system are separated into logical groups, so-called virtual hosts. In order to perform any action, a client needs permission to access the virtual host it intends to act within. Secondly, the client needs to be permitted to perform the particular action. RabbitMQ differentiates between three different types of actions: Configure operations add, remove, or alter resources within the virtual host. Write operations send messages, and read operations fetch them. The client needs to be allowed to perform the particular type of action on the particular virtual host in order for the action to be executed. This authorisation scheme is able to be exchanged using plugins to fit the particular need of the application [11].

Kafka communicates data in plaintext, i.e. in a highly vulnerable way. This means that confidentiality is not handled well by Kafka by default but it can be improved by implementing certain protocols. A way to achieve confidentiality would be to implement SSL authentication, as described in Section 3.3.4. If communication is done over both SSL encryption and authentication, then only the desired components or client applications, having the private key to decrypt the data, will be able to access the data [2].

3.3.2 Integrity

Integrity is the property of a system to assure the intended delivery of data or services [9]. Data corruption can occur on many different levels in an event-driven architecture. It can happen while processing an event, during transmission of the event notification, or during consumption.

The EDA should accommodate measures to provide end-to-end data integrity and lack of errors, as wrong data can have strong negative impacts in critical applications.

A widespread and easy to implement measure to reduce the error in the transmission phase is to use communication protocols that check the integrity of the data, like TCP [13]. This usually also has the benefit of increasing the reliability of the data transmission, as discussed in Section 3.1.

For the period of event processing that is unrelated to the pure transmission, more sophisticated measures against unauthorised access and unwanted data modification need to be taken, like proper authorisation of participants, and thorough testing of the event processing procedures.

In RabbitMQ, the use of TCP communication ensures that the integrity of messages is preserved on the network layer [11]. Proper authentication and authorisation techniques, as discussed in Section 3.3.4 and 3.3.1, can be employed to prevent the alteration of messages, whether they happen by accident or with malicious intent. Same goes for Kafka [2].

3.3.3 Accountability

Accountability means how well the actions of some entity can be traced back uniquely to that entity [9].

In the EDA world, accountability becomes very useful if there is a system error and you need to trace the module, be it a producer or a consumer, where the error has actually taken place.

Another use-case could be a system that contains a number of sensors as producers. A faulty sensor might start sending faulty events and therefore corrupting the message transmission handler. An accountable system would help with finding out the culprit so that the time to find the faulty producers out of the total number of active producers is reduced. This is especially important in very large-scale event-driven systems.

However, due to the strongly decoupled nature of most event-driven architectures, the participants are inherently anonymised. This may pose challenges with providing accountability in such a system, as the ability to identify other participants would mean stronger coupling, and thus weaker scalability.

RabbitMQ offers the so-called firehose tracer, which adds an exchange where all publishing and delivery notifications are also delivered to, next to their normal intended destination. This feature is not turned on by default, as it obviously comes at the cost of some performance and is mostly intended for debugging and development purposes. However, it does increase the accountability of the system considerably. Without this feature, the inherent decoupling of the system makes retracing messages a somewhat tedious affair of going through logged data [11].

Kafka, as mentioned above, provides distributed changelogs/commit logs that store all state changes and event or data related logs. This use of distributed changelogs helps increase accountability as every single change is stored in these changelogs which can be traced back when required [2].

3.3.4 Authenticity

Authenticity means that the identity of a subject or resource can be verified to be the one claimed [9].

An EDA designed with authenticity in mind makes sure that the events and messages consumed by the consumers were produced from trusted, verified, and authentic producers. This is important to ensure that there is no information coming from malicious entities and that the system has not been compromised.

But as with accountability, authenticity also suffers from the anonymisation caused by strong decoupling, and thus catering to authenticity can mean sacrificing scalability.

RabbitMQ offers secure authentication without stronger coupling of participants by requiring either username and password for a client to connect to the system, or a X.509 certificate. For properly verifying the password credentials, RabbitMQ enables the use of different authentication backends, which can be provided by plugins. An example would be the LDAP (Lightweight Directory Access Protocol) plugin, which comes pre-installed with RabbitMQ and offers both authentication and authorisation services using an external LDAP server [11].

By default, Kafka does not provide authentication. However, there is SSL (Secure Sockets Layer) authentication and encryption available for Kafka systems. Only adding SSL encryption makes it a 1-way authentication where the client application authenticates the server. To make it a 2-way authentication, SSL authentication also needs to be implemented which will make the broker authenticate the client as well. But there are systems other than SSL that can be used for authentication in Kafka [2].

4. CONCLUSION

Event-driven software is a popular choice for many modern applications in need of the ability to react quickly and asynchronously to changes in their dynamic environment, while being highly scalable. In this paper we presented a quality model for messaging systems employed in event-driven architectures, consisting of the main attributes of efficiency, reliability, and security, each with their own, more specific subattributes. We have also demonstrated where the attributes contradict each other, meaning that there is a trade-off between characteristics and a balance needs to be found depending on the application. The quality of software based on an event-driven architecture can be improved considerably when this model, its quality attributes, and the relationship between them are taken into consideration.

Future work that could build on this quality model would be the addition of quality metrics for the existing attributes, as well as internal quality attributes like the maintainability and portability of a messaging system.

Another avenue for further research are software patterns that are closely related to event-driven architectures, like for example complex event processing (CEP). Here the emphasis could lie on how to improve messaging systems for this particular type of architecture, with more directly applicable measures for improving the quality of the software.

5. REFERENCES

[1] S. Appel, K. Sachs, and A. Buchmann. Quality of service in event-based systems. In *Proceedings of the 22. GI-Workshop on Foundations of Databases, GvD*, 2010.

[2] Confluent, Inc. Confluent.io Apache Kafka Documentation, 2021.

[3] A. Corsaro et al. Quality of service in publish/subscribe middleware. *Global Data Management*, 19(20):1–22, 2006.

[4] S. Dixit and M. Madhu. Distributing messages using Rabbitmq with advanced message exchanges. *International Journal of Research Studies in Computer Science and Engineering (IJRSCSE), 6 (2)*, pages 24–28, 2019.

[5] P. Dobbelaere and K. S. Esmaili. Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper. In *Proceedings of the 11th ACM international conference on distributed and event-based systems*, pages 227–238, 2017.

[6] F. D. T. e Silva. Kafka: Ordering guarantees, Feb 2018.

[7] G. Fu, Y. Zhang, and G. Yu. A fair comparison of message queuing systems. *IEEE Access*, 2020.

[8] V. M. Ionescu. The analysis of the performance of rabbitmq and activemq. In *2015 14th RoEduNet International Conference-Networking in Education and Research (RoEduNet NER)*, pages 132–137. IEEE, 2015.

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

[10] B. Koldehofe. Principles of building scalable and robust event-based systems. 2019.

[11] Pivotal Software. RabbitMQ Documentation, 2021.

[12] J. Postel et al. User datagram protocol. 1980.

[13] J. Postel et al. Transmission control protocol. 1981.

[14] K. Sachs. *Performance modeling and benchmarking of event-based systems*. Sierke, 2011.

[15] K. Sachs, S. Kounev, and A. Buchmann. Performance modeling and analysis of message-oriented event-driven systems. *Software & Systems Modeling*, 12(4):705–729, 2013.

Software Testing Pyramid: How Architecture Influences The Shape

Sahner Pascal René
RWTH Aachen University
Pascal.Sahner@rwth-aachen.de

Kazantzi Maria
RWTH Aachen University
Maria.Kazantzi@rwth-aachen.de

ABSTRACT

Testing has been and will continue to be an important factor in the development of modern software. Many different strategies have been developed and hierarchically categorized. The test pyramid by Mike Cohn was one of the first approaches to quantify the amount of tests on each hierarchical level. Nevertheless, it is still debated whether the shape of a pyramid is the most suitable in general. In this paper we will introduce and explain several new test shapes and analyze them in dependence of the underlying software architecture.

Keywords

Software Testing, Test Shapes, Test Pyramid, Microservices

1. INTRODUCTION

As the demand for quality of software and speed of development rise, the need of reliable software testing strategies and efficient test development workflows does as well. With an increasing modularity of software, testing does also have to be diversified. The earliest commonly known approach to do so is the software testing pyramid by Mike Cohn [4]. From bottom to top it is divided into three levels: Unit, Service and UI. The width of each level represents the suggested amount of test, whereat the position determines the cost and effort for each test. Higher position imply increased cost and effort.
But soon a discussion arose: Some argue about the generality of the pyramid and some about its shape. One could even argue to add a third dimension as goals like speed of development, security, dependability and reliability have not been considered at all. It became clear, that the last word has not been spoken and some more refinement is needed to confront problems in different software architectures, environments and development processes.
In this paper we will explain the most common testing strategies and discuss several different shapes of the testing pyramid mainly in dependence of the chosen architecture. We will also scratch some possibilities to alter or refine the pyramid according to the corresponding conditions. As microservice architecture is increasingly used and very common today, we prioritize it in terms of depth of analysis.

2. THE IMPORTANCE OF AUTOMATION IN SOFTWARE TESTING

Traditionally, software was tested manually. For this purpose, it was deployed to a test environment and tested there by hand, e.g. by operating an interface. Usually, one proceeded along a document that gave instructions on how to perform the tests to ensure consistency. It should quickly become obvious that this way of testing is very time-consuming and error-prone: The one major source of error is that only black box tests are performed. White box tests can not be performed because it is not possible for the manual tester to analyze the control flow of the program. He can only operate on given interfaces. The other is of a social nature, because the monotony of such a test procedure causes the concentration of the testers to drop. This is the point at which automation becomes important. On the one hand, a wider range of test strategies becomes practicable - including white box tests in particular - and on the other hand, the susceptibility to errors is reduced. In addition, the developer receives feedback more quickly.

3. SOFTWARE TEST SHAPES

Before we can talk about the different layers of the testing pyramid more in depth, we first need to clarify the most basic terminology and ideas of it. As the terminology differs from paper to paper, we will give definitions for each one of the layers, but also like to point out, that there are correct interpretations aside from the ones given here. As depicted above, the pyramid is subdivided into three layers. They are briefly introduced and then explained, following the pyramid from bottom to top.

Unit Tests

Tests on the unit testing layer aim – as the name indicates – to test single units of software. [5] A unit can hereby be seen as the smallest part of software that represents a single most granular functionality. In the object oriented case these are usually methods and sometimes classes. Unit tests are meant to only test the single unit they correspond to. As the correctness of the underlying unit is mainly determined by its input and output behavior, these tests can be automated with little effort using testing frameworks. Predefined input sets are therefore run against the interface of the unit and the output is compared to a predefined output.

Integration Tests

On the layer of integration, the tests are meant to test the integration between the application and its environment. At the point of integration testing it is valid to assume, that everything already tested with a unit test works as expected. Those integration points can be file accesses, database accesses, network messages and API calls to just name a few. In these cases automation is state of the art as well, but due to their nature, their execution times tend to be longer than unit tests. Ham Vocke, an experienced software developer and consultant at Thoughtworks in Germany wrote in his blog: "They test the integration of your application with all the parts that live outside of your application"[17]. Integration tests are also referred to as "service tests" in other works, but we stick to the term integration test to draw a clear line to architectural components that are called service as well.

UI Tests

The topmost layer of the pyramid is called UI layer. The goal of UI tests is to test the user interface of the entire application. This must not be graphical ones. A command line interface is a user interface as well. Furthermore network based interfaces, such as REST APIs also count as user interfaces. As these tests cannot be automated in every case, testing can be slow and therefore cost-intensive. One example for such an unautomatable test scenario is the check if a graphical web interface looks "good" in all use cases. As it is even hard to formalize what a good look is, it is even harder to automatically check if the layout is broken from a graphical point of view.

3.1 Shape: Pyramid

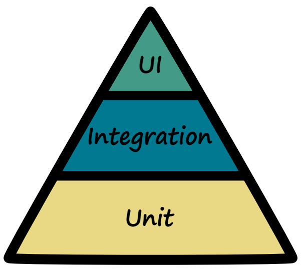


Figure 1: A pyramid shaped hierarchy

Without any information about the system that needs to be tested, the most obvious approach is to have a large set of unit tests as they can be implemented even during development, give fast feedback and ensure the most basic features to run as expected right from the beginning. Then use a smaller set of integration tests as the number of integration points is usually significantly smaller than the number of units. And lastly a minimal set of UI tests to reduce costs caused by manual testing. This results in the pyramid shape introduced by Mike Cohn [4] 1.

3.2 Shape: Hourglass

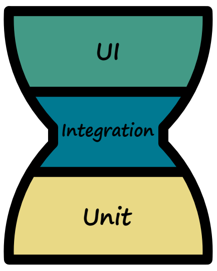


Figure 2: An hourglass shaped hierarchy

The hourglass describes testing scenarios with numerous unit tests, just a few or even no integration tests and a large number of UI tests. This kind of shape usually occurs when the application does not have many integration points. In those cases it is important to pay much attention to the large UI testing layer as this can potentially become cost-intensive.

3.3 Shape: Pencil

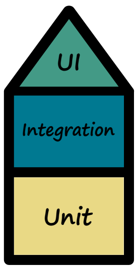


Figure 3: A pencil shaped hierarchy

The pencil shape is used to refer to scenarios where the amount of unit and integration tests is nearly balanced and the number of UI tests is minimized. Systems with a strongly subdivided structure can fall into this schema as the integrative part becomes larger and thus requires more tests. It is also possible that the amount of unit tests is decreased, as some functionality is represented by the integration of multiple modules. This shape was introduced by Simon Brown [2] but not named.

3.4 Shape: Diamond

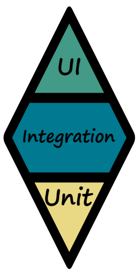


Figure 4: A diamond shaped hierarchy

The diamond shape is used to describe scenarios where only the integration layer plays a greater role in testing. Unit and UI tests are mostly avoided. Projects composed of many stable subsystems may fall under this category. As the single working subsystems do not need to be tested, the unit test part is not needed. The integration of the working modules has to be tested then and the number of UI tests is kept low as they are usually the most expensive ones. This shape was mentioned in a blog written by Mateusz Roth [16].

3.5 Shape: Ice Cream Cone

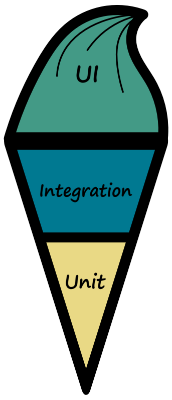


Figure 5: An ice cream cone shaped hierarchy

The ice cream cone shape is often referred to as an anti-pattern, but it is not necessarily one. It uses a minimal amount of unit tests, just some integration tests and mainly focuses on UI tests. This has the potential to become costly and inefficient as it relies mainly on UI tests. In addition, if a test on the UI layer detects an error on the unit layer it might be hard to find the cause. This shape can also be found in Mateusz Roths blogpost [16].

4. COMMON TESTING STRATEGIES

4.1 Unit Layer

On the unit test level there are three strategies commonly in use.

4.1.1 Black Box Testing

Firstly, testing software in black box fashion is a technique, where the code or internal logic is not known or not of interest. Due to the limitations of not knowing the internal structure, the defined test sets can only be created by the specification and requirements regarding the unit.

Some commonly used techniques in black box testing are the boundary value analysis (BVA) and the equivalence partitioning testing (ECP). In BVA the tester focuses on inputs which are the boundaries of possible values or edge cases in computation. Using the ECP testing technique, one partitions the input set in consideration of the input and output values. This reduces the overall number of input values as one for each class is sufficient. This has the advantage of reducing the needed test cases and therefore speeding the testing up. Comparing these two techniques, they have a lot of parallels as they both try to reduce the amount of tests needed. The inputs of the BVA can be seen as classes in the ECP. The classes of the ECP can be seen as boundary values in turn as they each contain only a single representative. The BVA should be used when a special focus is set on detecting errors in special cases. The ECP should therefore be used when the input is easily classifiable [10].

4.1.2 White Box Testing

The second common strategy of unit test is white box testing. In comparison to black box testing, the inner structure of the unit or its source code is known. This makes it possible to analyze the control flow during the execution of tests. Branches that are strongly dependent on the implementation and not the specification, would possibly not be considered in black box testing. It is widely practiced to measure the quality of white box tests by checking their line coverage. When trying to improve the tests quality, it is important to ensure that at least every non trivial line is tested.

The control flow testing is used to determine the execution order of instructions. This is usually done by modelling branches, loops and instructions as a directed graph and checking whether the order of visited vertecies is correct. The term path coverage is used to describe how many paths beginning in the start vertex and ending in the end vertex are tested. As this can be infinitly many in case of the existance of a cycle, only the most important ones should be considered. Another known technique in white box testing is the basis path testing. Here the test cases are based on logical paths or flows that can be taken through the program. It makes sure that each path of the code is taken in a predetemined order. The tester executes all possible blocks in a program with the aim to achieve maximum path coverage with the minimum number of test cases. This technique helps to have test cases which execute every statement in the program at least once.

In addition, using white box tests, the intermediate values of computations can be accessed. This is useful to analyze in more detail. White box testing is in general slower as more tests are required to take advantage of the knowledge of the unit. This also means, writing these tests takes more time.

4.1.3 *Gray Box Testing*

In order to overcome this problem, there is a mixture of the two called gray box testing. In this test the internal structure is at least partially known. The tester has enough access to the algorithms and structures to be able to design the test cases, but the testing itself is on black box level. The big advandtage is that, even knowing the structure, the design process of tests is speed up as the granularity of test cases is reduced in comparison to white box tests. But as the structure is known the input sets are designed to fit the unit and cover its most important parts.

One example for a strategy using gray box testing is the regression testing. Here the tester makes a functional improvement or fixes an error in the program. The aim of this test is to determine whether that change has influenced other parts of the program, by running new test cases. This technique is used to confirm that the product is improving in terms of quality and not relocating the faults by causing new ones [1].

Comparing these three strategies, it is observed that black box testing does not consume as much time during development as gray and white box testing. This is because the structure of the underlying unit is not considered and complexity is left out. When putting a little more resources into testing, gray box testing is suitable. It makes use of the knowledge of the structure of the unit, but is not as detailed as white box testing. As white box testing can be exhaustive and time consuming, it is used for important parts of the system as it more likely to detect bugs and errors [1].

4.2 Integration Layer

The best known testing techniques on integration level are the top down integration and its opposite technique, bottom-up. The top down technique is used to test all the modules from top to bottom. Firstly, the higher levels are integrated by moving downward through the structure, beginning with the main module. Then the next modules are integrated to the main module and are incorporated into the structure in either a breadth-first or a depth-first manner. This process has to be repeated automatically in order to test all the modules together as a unit. One disadvantage of this methode, is that the modules at a lower lever will not be tested enough. The bottom-up integration has the reversed way of working from the top-bottom technique. Unlike top down Integration, the modules that must have a priority are tested in the end.

4.3 UI Layer

Recovery, security and compatibility testing are some techniques that are being used in the system testing level. During recovery testing, the tester tries to fail the software with an eye on verification of the recovery of software. It is important to see, whether the software is able to work properly after such an error. Secondly, the security testing is a technique to find all system problems in an application. The tester behaves like a user who is going to use the system. That means he will try to overwhelm the system, probably cause a lot of system errors and browse into insecure data. In that way the security of the system will be tested. Advantage of this technique is that it is fast and accurate. However, it is not possible for the tester to cover all the security vulnerabilities and cases. The last technique, compatibility testing is focused to evaluate the compatibility of the system with the computing environment. Herewith the software will be installed on different operating hardware systems and networks. This technique and the recovery strategy are time consuming and slow.

5. RELATED WORK

There are many blogs analyzing the test pyramid regarding microservices, but the focus is rarely laid on the shape. [3] argues for a diamond shape. In his figure, the layers are exactly upside down in relation to their usual order. But considering the symmetric properties of a diamond shape, this does not play any role. [13] also argues for a diamond shape. He additionally introduces a new layer below the unit testing layer: The static layer. It is meant for tests using static elements of a language. The type check in javascript is given as an example. [16] also addresses problems regarding the shape. He argues for a trophy shape when considering static tests and a diamond shape otherwise.

6. SHAPE ANALYSIS

Now as we clarified terms and gave cummon knowledge of testing with the pyramid, it is possible to start with the architecture dependant analysis.

6.1 Monolithic Architecture

Monolithic architecture is also called multi-tier architecture due to its internal structure. It usually consists of into three layers: The presentation layer, the application layer and the data layer.
At first, the presentation layer is the view of the system from a users point of view. This is also called the user interface. The presentation layer communicates with the other two by using internal API calls. Next, the application layer handles HTTP requests from the interface. The Data layer uses to manage all queries comming from the application layer and coordinates data accesses. [11]. Characteristic for monolithic systems is, that it includes multiple services in a single program from a single platform. This causes the code for each unit to be deployed at the same time. Those different services communicate with each other using external systems or consumers from different interfaces. The application runs independent from other computing applications and if a developer intends to update parts of it, he can make changes to the whole stack at once.

6.1.1 *Advantages*

This architecture in general has several advantages:

1. The Monolithic architecture contains all of its modules in a single deployable unit.

2. It is simple to test owing to the fact that the tester is

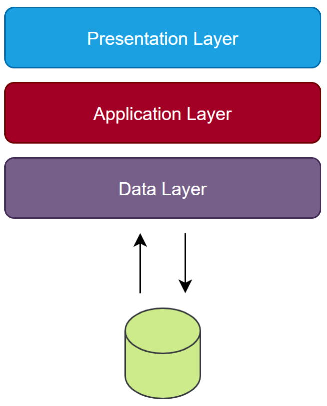


Figure 6: The structure of a monolith architecture.[12]

able to easily create end-to-end test case by running the application manually.

3. It is easy to deploy, because it is a single unit.
4. At the beginning of a project it is easier to develop within a monolithic architecture, since it is small and manageable.
5. It is easier for developers who did not participate in the development right from the start to understand how the application works, because it has only one code base.

6.1.2 *Disadvantages*

On the other hand when a product grows in size, the monolithic architecture is not the most efficient structure for it.:

1. The structure grows more and more complex.
2. The level of complexity rises as a result.
3. It becomes harder to manage and test the different modules, because they have to be tested at once.
4. Cooperation between different teams requires more effort as they develop in the same environment and have to pay attention not to do any contradicting changes.
5. When the application is already developed, the team can not easily change the framework and the language that was used. Even if they only look at a single module.
6. The need to change a library for a certain module might destroy the functionality of another module.
7. In this way the development slows down.
8. Load balancing and scaling is almost impossible at a certain size.
9. Malfunction of single modules cause the entire system to fail.
10. Even for minor changes the whole system has to be deployed again.

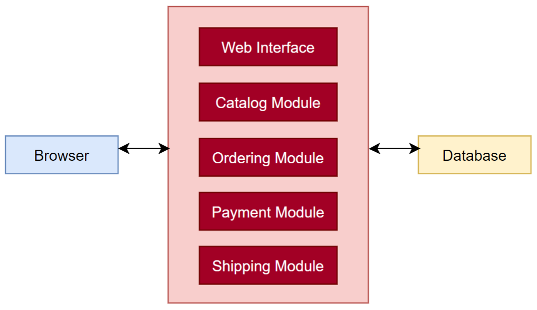


Figure 7: Example of a monolithic architecture.

6.1.3 *An Example*

Consider the example given in figure 7 of a monolithic application. This application contains a web interface, a catalog, ordering, payment and shipping module. All these modules should use the same file system. They are deployed as one unit. When testing this application, the focus should lay on the business logic and communication between modules. On the unit layer one would use white box and black box tests to test modules. In this example that would be the catalog, ordering, payment and shipping module. Integration testing is then used to check the interfaces between the modules. With special respect to the payment module as its functionality is of great importance. Moreover, regression tests can be applied to verify that changes in the code will not effect the functionality of the system. Apart from the functionality, the security of the application has to be tested as well. This can be achieved by using end to end tests. End to end testing involves testing an applications flow from beginning to end. Testing tools help to provide flexibility and to use test cases that have been used before, due to changed specifications. Furthermore it makes it possible to have results and status at the time of the test. Testing the web interface would fall under unit testing.

6.1.4 *Shape Analysis*

Literature research did not yield any results setting the pyramid into relation to testing practices regarding monolithic applications. Especially no emperical data was found. The research was finished June 13, 2021. So in the following we will just argue about the shape.

If there is a lot of business logic, the number of unit tests is probable to be large. And this is usually the case as single modules are large, because they have to cover a lot of functionalities. In relation, the amount of integration tests should not be larger as monolithic applications usually do not have many integration points. As it is shown in figure 5 and figure 4 the number of unit tests is smaller in comparison to the number of integration tests, which is not the most advantageous approach in the case of monolithic architecture. Therefore the ice cream cone shape 5 and the diamond shape 4 can be excluded. When the application has to communicate a lot with external systems or is strongly subdevided into modules, the integration layer can grow in size. This would make a pencil shape 3 possible. Furthermore, limiting the amount of UI tests is rewarding as they are the most expensive and have long runingtimes. This is caused by the fact that every request has to go through all layers. Taken into account all this information, a pyramid shape seems to be the most suitable. A pencil shape is rather unlikely, but possible. The last case we look at is a large application only consisting of a handfull modules. In this case the complexity is moved from the integration to the UI layer. The unit layer is excluded in this shift as those tests should not test communication between units by definition. An hourglass 2 is then the result.

All in all, a pyramid shape is the most likely, the pencil shape and the hourglass shape are possible.

6.2 Service Oriented Architecture

The original idea for service oriented architecture comes directly from the scalability and maintainability problems of large monoliths. The aim of service oriented architecture is to break down monoliths into smaller services. Even though these services mostly share one data source, they are much less coupled in contrast to the monolith. Nevertheless, they communicate together via a single enterprise service bus. If one service causes problems with the bus,

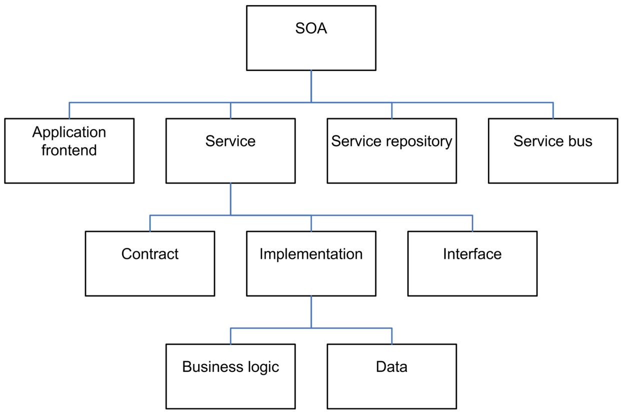


Figure 8: The structure of soa architecture. Source: [6]

this can affect others. In contrast, integration points to other applications and services can be implemented much easier. If we now compare the service-oriented architecture with microservice architecture, the services are much more extensive and stronger coupled. Figure 8 gives an overview over the structure of a service oriented application.

It should be noted that this architecture has been in use for a long time: It was used to a greater extent from around 2000, but was largely superseded by the microservice architecture since around 2010. The test pyramid was first mentioned in 2009. Therefore, the work used here has no references to the test pyramid. Let us now make an initial assumption:

We first take a look at the individual service: By design, this should be only a part of the overall system, but in itself it can also grow to the size of a monolith [12]. For this we have already established the pyramid 1 and the pencil 3 shape as the most likely forms in the previous section. We now take this as the basis for the set of individual services. Since the user interface should be independent of the underlying architecture, we also assume that the number of UI tests in the service oriented architecture is equal to that of the monolith. However, we now add the enterprise service bus and the service repository as integration points. This increases the integration level of the shapes to a diamond shape 4. Since we did not find any empirically proven studies, statements and assertions are quoted in the following and logically linked in order to argue about different shapes. The Paper [15] points out that unit testing behaves in the case of the service oriented architecture the same, as with usual software components. Therefore a proportional connection between the quantity of business logic and the number of unit tests can be concluded directly. If an application has thus much business logic, then also the need for unit tests turns out to be high. Furthermore, the services can be tested as whole as black boxes. Test cases can even be generated directly from the specification of the service [15].

On the integration level, the behavior of the services among each other is tested. Tests regarding application external sources can also be added. In particular, these sources must be tested to determine whether their behavior has changed and whether they still react as expected.

On the integration layer, the non-availability of services does also play a role[15]. The overall picture now shows that the number of integration points is decisive for the number of integration tests. This in turn depends on the number of services on the one hand and on the number of external interfaces and dependencies on the other. A large number of services and many external interfaces therefore increase the number of integration tests required.

In 2007 there was no solution for automated UI testing for service oriented applications[15]. Monitoring the running system is mentioned as a solution. By now, these frameworks exist. As an example SoapUI is given `https://www.soapui.org/`. Nevertheless, since these tests are particularly cost-intensive and require a running instance of the entire system, they should be used less frequently in relation to the other test types. All in all, the resulting shapes can be roughly narrowed down: The ice cream cone shape falls out of the grid, since the number of integration tests alone should be greater than that of the UI tests. The same applies to the hourglass shape. The scope of the business logic and the degree of subdivision, i.e., the number of services, remain as variables. If a system consists of only a few services, whose nature is more like that of a monolith, a pyramid shape is created. Provided, of course that the main task of this system is not communication, i.e. integration with external components. Then the distribution of the tests would look more like a pencil shape. This is also the case if the scope of the business logic remains the same, but the subdivision into services is more granular. The integration points created in this way then have a direct effect in the form of an increase in the size of the integration level. In case the amount of business logic is small, a weak subdivision into services results in a pencil shape again. If the degree of subdivision is large, however, a diamond shape 4 can also result.

6.3 Microservice Architecture

Modern, web-based applications are increasingly being developed and deployed in the microservice architecture style. In this style, the individual functionalities of the overall system are divided into microservices that are as independent as possible. The microservice architecture can be seen as an implementation of the software oriented architecture. The particular difference is that a special focus is set on designing the services as minimally as possible. In addition, independance and loose coupling of the services are desired properties. Microservices can therefore be regarded as a separate architecture [9]. Graphic 9 shows the structure of

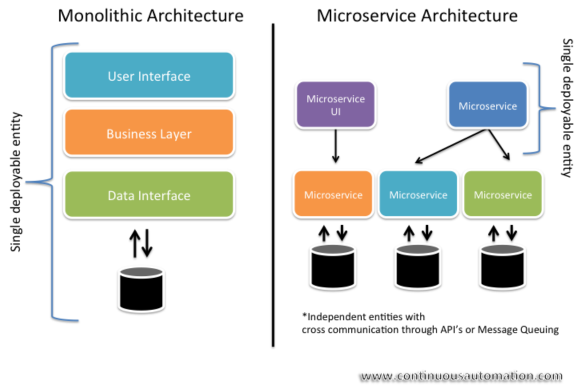


Figure 9: The structure of a microservice compared to a monolith.[12]

such a system. Advantages of this architecture are seen in particular in the fact that the modularity increases and individual components can be changed independently of one another or even exchanged, provided that the interfaces do not change. In general, the goal is to have to make only minor changes to the overall system in case of minor changes to requirements, technology or the overall scope of functions. Loose coupling is particularly advantageous in the development process: Each service can be assigned to a development team. This allows the teams to develop more independently of each other. Even in different programming languages and using different technologies. Companies in particular see this as a way to cut costs and work more efficiently, since unnecessary delays between teams are reduced to a minimum. In addition, these systems are easier to scale than monoliths: services can be developed individually so that they can be instantiated multiple times. On the other hand disadvantages include network latency and the possible failure of communication. However, the greatest difficulties arise especially when it becomes necessary to change the interfaces of services in cause of further development. In addition, there are all the problems that distributed systems bring with them in contrast to monoliths.

6.3.1 *Refining The Integration Layer*

Since the most complexity now lies at the network level, a special look must be taken at the integration layer of the pyramid. For this purpose, it is divided into two layers: On the first of those layers the contract tests are categorized. These are used to check the correctness of an interface. For this purpose, the services or its components are first divided into producer and consumer. A consumer is defined by the fact that it accesses resources provided by external sources. A producer, on the other hand, provides resources to external consumers. Contract tests can now be defined from producer or consumer side. These are then referred to as consumer-driven contract tests or provider-driven contract tests. In consumer-driven contract testing, consumers define their expectations for the interface they consume. In practice, this type of testing has the advantage that in the early stages of development, the producer team can implement along the expectations specified by the tests, implementing only the most necessary functionality. As long as the tests continue to run without errors, any changes can be made. This gives development teams more autonomy. [17] Provider-driven tests should be implemented by the other team. That is by those who provide a resource. They then have to use the tests to check whether they violate a previously defined property. This is usually not possible with public APIs, since not every consumer can be taken into account. Otherwise, the development process would stagnate. Within a company, on the other hand, it is possible and practiced[17]. Both variants thus test the correctness of the provided interface. As already mentioned, the interfaces and their consistency and definition are decisive for the functionality of the overall system. Therefore, contract tests are used to determine whether the service as a whole responds correctly and in a predefined way to external requests.
The cases in which a microservice acts as a consumer are assigned to the second layer. It is called the same as the layer before splitting: Integration layer. The integration tests associated with it are intended to test whether the service correctly accesses the resources it needs. However, this also includes operating system resources, file system accesses and all other components already localized at the integration layer in monoliths.

6.3.2 *Shape Assumption*

The part of the application known as business logic is tested with unit tests, as in the monolith. This part should be minimal according to the concept of the architecture.

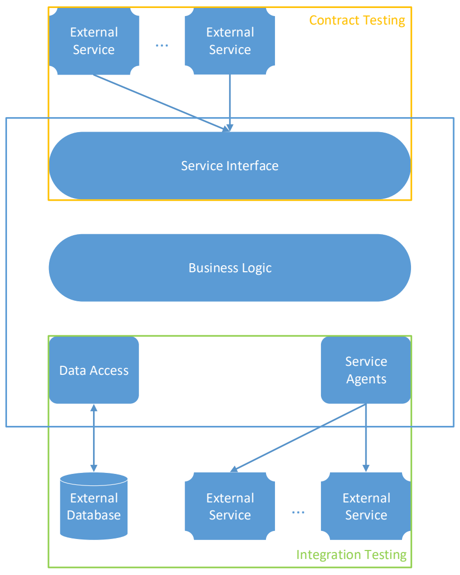


Figure 10: The structure of a microservice with a mapping to testing layers. Adepted from [14]

Figure 10 visualizes the mapping components of a microservice to the layers. If the entire system consisting of

all services is tested at once, this is called an end-to-end test. This type of test is usually classified as UI test. This is how Mike Cohn suggested it[4]. It is also important to note that exposed interfaces (e.g., REST APIs) count as user interfaces, just like CLIs and web interfaces do. Thus, they also fall under the category of UI testing. Even if testing in the case of REST APIs can be done using integration testing methods and tools, these are to be located at the level of UI tests. The following assumption arises from these procedures, which are taken from common practice: Since the services are usually minimal, the number of unit tests is reduced. The communication that such a system would perform on the module level if it was a monolith is shifted to the network level. This logically increases the number of integration points. It also increases the number of integration tests required. Since the system does not change from the point of view of a user and UI tests are generally more disadvantageous than the other test types, their quantity does not change.
Now, if we follow the usual testing procedures, the pencil shape turns out to be representative for single services. Should the overall system consist of multiple microservices whose test shape corresponds to the pencil shape, it also results in a pencil shape for the entire system.

6.3.3 Shape Analysis

In the following section, we will analyze whether this assumption can also be proven empirically or at least on a case-by-case basis.

In the online journal “Yo Briefcase”, James Hughes argues in his article ”Micro Service Architecture” [8] for a general reduction in the number of tests. In particular, he emphasizes that tests should be as much value adding as possible. His argument is based on the principle that a microservice should be responsible for only one functionality, but do it well. From this he concludes that the general risk of introducing an error through changes, which leads to the dysfunctionality of the overall system, is reduced. He also names the “natural behavior of a service based system” as evidence. From this point of view, it cannot be clearly concluded at which levels the reduction of tests should be classified compared to the monolith. But with the presented view on microservices “do one thing, and do it well”, it is probable that the unit layer is meant. This would speak for the pencil shape. On the other hand, the reduction of the tests can also be seen as equal. This would only shrink the pyramid, but not change its shape. An ice cream cone cannot be inferred with this reasoning, since nothing explicit was said about the UI tests.

Simon Brown comes to a more precise conclusion in his blogpost “Modularity and testability” [2]. However, it should first be noted that in this article he is primarily compared modularized monoliths and microservice architecture. He claims that when the modules of the monolith are tested as black boxes, the shape of the pyramid changes. He describes the changed state as “a balanced mix of low-level class and higher-level service tests”. In addition, he states that microservices are likely to fall into the same grid. In the image he presents for this, the Pencil shape is even depicted 11, although it is not referred to as one. However, he then restricts the statement again: there is no typical form of the

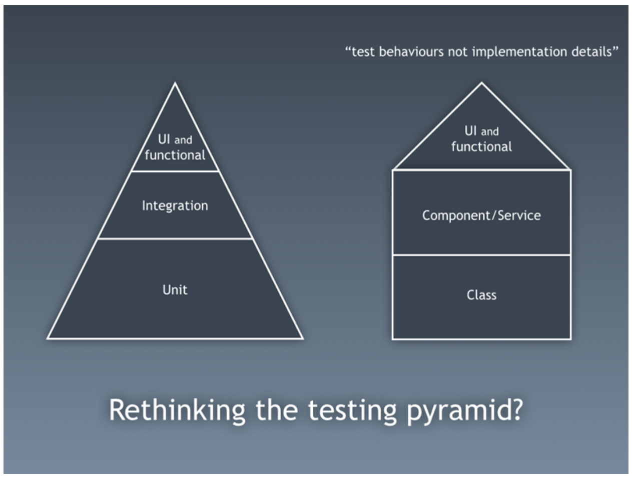

“test behaviours not implementation details”
UI and functional
Integration
Unit
UI and functional
Component/Service
Class
Rethinking the testing pyramid?

Figure 11: The shape presented by [2]

testing pyramid. The form depends on the concrete project. How the project now influences the form, he does not write. This reinforces the assumption made at the beginning that in general a pencil shape applies to microservice architecture.

However, these are not empirically proven claims. The meta-study “Testing Microservices Architecture-Based Applications: A Systematic Mapping Study” [18] looks more closely at microservices testing. For this purpose, 2481 studies were pre-selected and 33 of them were chosen after four steps of selection. The distribution of the studies regarding their types can be taken from figure 12, the distribution over the methodology from figure 13. The distribution over

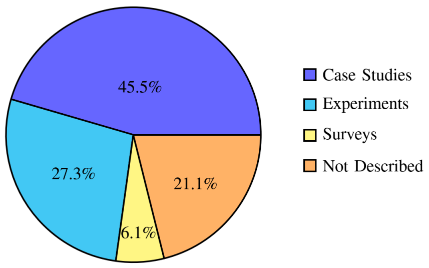


Figure 12: The research methods used in the primary study [18]

the years can be taken from figure 14. The first and most important finding of the study is that unit and integration tests are the most popular approaches. The analyzed studies commonly agree in categorizing tests of the business logic unit tests. Let us now take a look at figure 16. It gives the usage number of each approach used in the primary studies. Interesting to note is that the importance of integration tests is given to be greater than the one of unit tests. Given that business logic testing is considered unit testing and the relation of testing approaches in 16 represents roughly the relation of functionalities in a generic microservice based

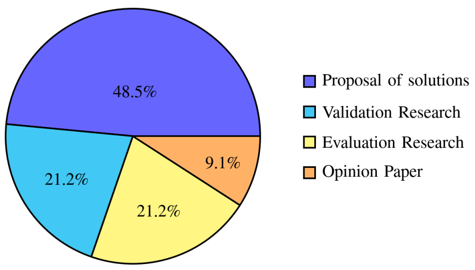


Figure 13: The type of the primary study [18]

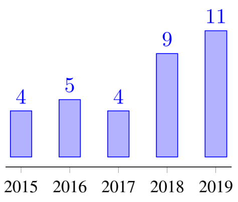


Figure 14: The years from that material was used in the primary study [18]

system, it is possible to conclude that in common the integrative part is larger than the unit part. This underlines the increased importance if integration tests in microservice environments. Furthermore, the importance of integration testing in the overall community picture is summarized as follows: "addressing the challenges in automated and intercommunication testing is gaining the interest of the community". From that we can conclude that integration testing is at least as important as unit testing and UI testing is done the least. This speaks clearly against a pyramid 1 and an hourglass 2 shape, as the integration layer is valued to small. The statement that unit and integration testing are the most important excludes the ice cream cone shape 5 as well. The overall shape extracted from this study is depicted in figure 15. This is still no contradiction to the assumption

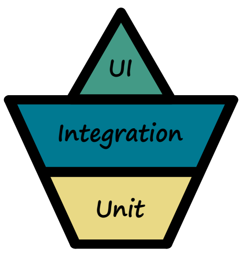


Figure 15: A UFO shaped hierarchy

of the pencil shape 3. It rather gives the information that the shape of testing microservice based applications mainly depends on the granularity of the single services. The larger services are, the larger the unit layer and the smaller the integration layer becomes, while the UI layer stays the same. In the case of maximal granularity a diamond shape 4 is representative, where low granularity and large services state for a pencil shape 3.

To sum up for microservices, it is not possible to limit the selection of shapes to a single one. Depending on the concrete application shapes from diamond to pencil are the most likely. Being close to a pyramid, ice cream cone or even hourglass shape is usually an indicator that either the focus of the tests can be refined or the application violates the definition of a microservice based system.

7. LIMITATIONS TO GENERALITY

The overall validity of the findings in this work depends on the actual need to tests specific components. If programmers and designers never made mistakes, testing would be obsolete. This is not the case indeed, but generated or reused software can achieve similar circumstances. This might be from greater importance in future, because for many projects the definition of models and structure is besides from the user interface enough to describe the entire system.
In the following we give an example of a microservice based application where the ice cream cone shape is optimal.
Consider a small company which produces and sells articles via an online portal. The portal has a product management, a user login and some way to pay. If we first define the models corresponding to users, this information is enough to derive the database models from it. If we add validity information to it, even the generation of a service between the database and the browser is possible, as it only receives request, checks and forwards them. Same applies for the product palette. When considering permission management, given user roles, it is possible to generate a service including its database for permission management and authentication. When using the same platform for generation, it is even possible to generate the authentication code on the integration layer.
The only thing missing is the user interface, as all API endpoints are generated. As this is very customer specific, we do not assume it to be generatable. As the unit level code is completely generated, no unit tests are required. Even on the integration level only the communication of the user interface with the endpoints has to be tested. What remains are tests of the user interface and further manual tests.
This would result in an ice cream cone shape.
The shape does also thinner on the lower two levels when functioning services are used as part of a new application. Testing this service then belongs to another project or company even.

8. CONCLUSION

As it turns out the architecture has a large influence on the shape. Depending mainly on the amount of business logic and granularity, the integration and unit layer vary in size. It is usually a good idea to keep the UI layer as small as possible independant from the architecture. For monoliths,

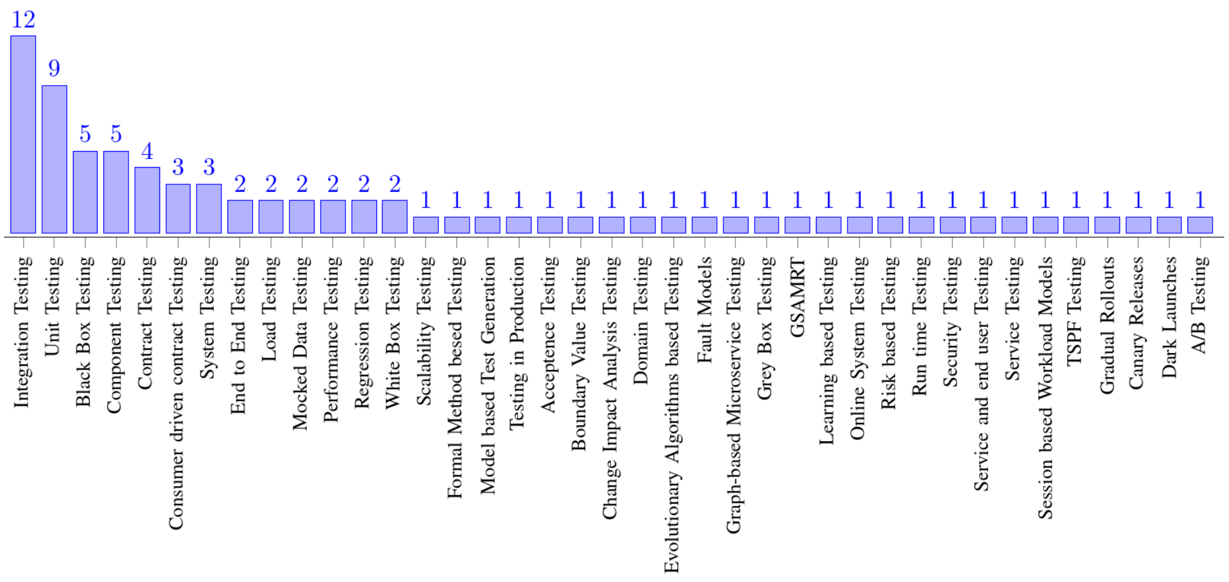


Figure 16: The testing approaches for microservice-based applications from the study [18]. Single approaches can be categorized multiple times.

a pyramid shape is the most likely, the pencil shape and the hourglass shape are possible. Using SOA a diamond and pencil are likely, a pyramid is possible. In a microservice environment, same arguments as for the SOA apply, but the granularity can be assumed as high and the size of a service as small. Therefore a pencil, diamond and ufo shape are common. We also pointed out giving an example that the ice cream cone is not necessairly an anti pattern. To fulfill future demands new test shapes will arise and new architectures will reqire more refinement. Although it is strongly project dependant.

9. REFERENCES

[1] S. Acharya and V. Pandya. Bridge between black box and white box–gray box testing technique. *International Journal of Electronics and Computer Science Engineering*, 2(1):175–185, 2012.

[2] S. Brown. Modularity and testability, Oct 2014.

[3] D. Cilano. Test automation - from ”test pyramid” to ”test diamond”, Jul 2019.

[4] M. Cohn. Succeeding with agile: Software development using scrum, addison-wesley. 2009.

[5] M. Cohn. The forgotten layer of the test automation pyramid, Sep 2017.

[6] W. Commons. Soa elements, 2006.

[7] M. Fowler and J. Levis. Microservices, Mar 2014.

[8] J. Hughes. Micro service architecture, Apr 2013.

[9] J. Huttunen et al. Micro service testing practices in public sector software projects. 2017.

[10] P. M. Jacob and M. Prasanna. A comparative analysis on black box testing strategies, 2016.

[11] M. F. James Lewis. Microservices. March 2014.

[12] J. McAllister. Microservices decoded: Best practices and stacks - dzone integration, Nov 2015.

[13] J. Nyman. Test shapes - stories from a software tester, Sep 2020.

[14] T. Pickens. Testing strategies for microservices, Nov 2015.

[15] L. Ribarov, I. Manova, and S. Ilieva. Testing in a service-oriented world. 2007.

[16] M. Roth. Why the test pyramid is a bullshit - guide to testing towards modern frontend and backend apps, Oct 2019.

[17] H. Vocke. The practical test pyramid, Feb 2018.

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