On Cloud Computing Infrastructure for Existing Code-Clone Detection Algorithms

Andrew Walker
Computer Science
ECS, Baylor University
One Bear Place #97141
Waco, TX 76798
andrew_walker2@baylor.edu

Tomas Cerny
Computer Science
ECS, Baylor University
One Bear Place #97141
Waco, TX 76798
tomas_cerny@baylor.edu

ABSTRACT

Microservice Architecture (MSA) is becoming a design standard for modern cloud-based software systems. However, even though cloud-based applications have been thoroughly explored with regards to networking, scalability, and decomposition of existing monolithic applications into MSA based applications, not much research has been done showing the viability of MSA in new problem domains. In this paper, we explore the application of MSA to the code-clone detection problem domain to identify any improvements that can be made over existing local code-clone detection applications. A fragment of source code that is identical or similar to another is a code-clone. Code-clones make it difficult to maintain applications as they create multiple points within the code that bugs must be fixed, new rules enforced, or design decisions imposed. As applications grow larger and larger, the pervasiveness of code-clones likewise grows. To face the code-clone related issues, many tools and algorithms have been proposed to find and document code-clones within an application. In this paper, we show that many improvements can be made by utilizing emerging cloud-based technologies.

CCS Concepts

•**Applied computing** → *Enterprise applications; Service-oriented architectures;* •**Software and its engineering** → **Formal software verification; Software maintenance tools; Software verification and validation;** *Parsers;*

Keywords

Microservices, Cloud Computing, Code Clone, Clone Detection, Scalable Code Clone Detection, Software as a Service

1. INTRODUCTION

Microservices [8, 14] are the latest trend in software design, development, and delivery. Several benefits are often associated with microservices, including faster delivery, improved scalability, and greater autonomy. Greater autonomy also provides features such as smaller codebases, strong component isolation, and organization around business capabilities. These benefits promise improved maintainability over traditional monoliths.

In this context, it is not surprising that demand has grown for migrating legacy monolith applications to microservices. The research in this area, which provides design patterns and guidelines on how to implement the migration, is substantial. However, most of these studies are from the macro architecture perspective, and they target issues such as identifying candidates for microservices on the monolithic system or separating these candidates into a hybrid architecture [14]. We wished instead to focus on the issue of transferring this cutting-edge technology to a new problem domain.

In this paper, we introduce a problem domain that we show can significantly benefit from the utilization of cloud technologies. Our chosen domain is code-clone detection algorithms and applications. To show how the problem domain can benefit from cloud technologies, we first explored the new domain and identified a series of four problems that should be addressed by future code-clone detection algorithms and applications. We then implement an existing code-clone detection algorithm, Sourcerer-CC [41], as a cloud-based application that we named Corpus-CC to show how a cloud-based solution can be used to solve the issues we identified. Beyond just resolving issues in the problem domain, cloud-based solutions offer the ability to extend the domain into new areas of research. Our tool is applied in one of these new areas of research, *inter*-application code-clone detection, and two experiments are run to show the viability.

The remaining content is organized as follows - Section 2 covers the background of microservice architecture, and Section 3 covers related work. Section 4 explores the new problem domain, including background, state-of-the-art, and problems within the domain. Section 5 introduces our proposed solution and gives results on two use cases. Lastly, Section 6 covers future work and concludes the paper.

Copyright is held by the authors. This work is based on an earlier work: RACS'19 Proceedings of the 2019 ACM Research in Adaptive and Convergent Systems, Copyright 2019 ACM 978-1-4503-6843-8. https://dl.acm.org/doi/10.1145/3338840.3355659

2. BACKGROUND

Monolithic architecture produces large systems that are deployed as atomic units, which makes them hard to evolve or update. When one component in such a system was modified, it usually meant extensive testing and redeployment of the entire structure. Also, scaling a single component meant scaling the whole application.

Microservice architecture aims to solve the challenges of monolithic systems. The main emphasis is on systems' modularity. Software built with MSA is composed of multiple component services. So each of them can be tweaked, updated, and deployed separately without compromising the integrity of the application. Therefore, the developers can update the system and redeploy just a single module instead of the whole app. From the business perspective, this also means that, instead of having different teams handling the back-end, front-end, operations, and quality assurance, each small team owns a microservice. In other words, the team not only creates it but also takes responsibility for deployment and maintenance.

Microservices are usually deployed in service containers like Docker [10]. An infrastructure like this requires an orchestration system to provide the necessary features such as automated deployment, scaling, security [47], service discovery, load balancing, or externalized configuration. There are open-source container orchestration solutions such as Kubernetes [32] or Docker Swarm Mode that can be used.

Each microservice defines an interface that other components can consume, and the services communicate via RESTful APIs or through a message broker. The message routing is simple. There is no centralized element integrating the services; the governance, as well as the data management, is distributed. This interaction style is called dumb pipes and smart endpoints.

Since each service uses a different data-store, there is no need to share the data model across the whole app (canonical data model). Instead, each service operates on a subset of the data model in a so-called *bounded context* [14]. Since each service specializes in a different business case, naturally, not all services need to operate with all entities. A service may even consider only certain attributes of some object and ignore others. For example, in some systems, a person management service uses all the information about users, including the degree they pursue. However, a hotel management service would not need the degree attributes at all.

3. RELATED WORK

The popularity of microservice architecture (MSA) has grown consistently over the past five years [5]. As an example of this, just look at the Google search trend for microservices during the last five years 1. During this time, many businesses migrated their systems from monolith or Service Oriented Architecture (SOA) into MSA [8]. MSA has become a core architecture concept for many big tech companies [18, 27, 34]. However, microservices aren't silver bullets, and the difficult process of migration has drawn the attention of the industry as well as academia.

Figure 1: Google search trends for microservices

Taibi et al. [45] conducted a study interviewing experienced developers and examined the motivations, issues, and benefits behind migration to MSA. The study concluded that the critical drivers for migration are the overall maintainability and scalability. However, the main technical issues were monolith decoupling, database migration, and data splitting.

Another empirical study with MSA practitioners [9] describes the incremental migration. The researchers decompose the process into three parts: reverse engineering - gaining knowledge about the legacy system; architecture transformation - domain decomposition and applying domain-driven design practices; and forward engineering - the actual implementation of the new system. They recommend adding new functionalities written as microservices and then incrementally migrate the existing ones.

Knoche and Hasselbring [26] also highlight maintainability as the critical reason supporting modernization. They argue that monolith systems are challenging to extend since any change requires extensive testing and rework. And that limiting the number of entry points and establishing platform-independent interfaces allows the future evolution of the system to become more feasible. They describe modernization from COBOL to Java, and, for complex high-value systems, they suggest first to define the service facades, then implement them in the legacy technology, and to finally re-implement them again as Java microservices.

In 2017, Balalaie et al. [5] published a catalog of migration and rearchitecting patterns derived from the observation of several industrial projects. They provided general guidelines with a concrete technology stack for implementing them.

One of the challenges with migration is the identification of the microservice candidates on the monolithic system. Levcovitz et al. [30] proposed a technique based on mapping the database tables on business areas and facades, which creates a dependency graph that can be used to identify the subsystems. In 2018, Zhongshan et al. [38] went even further with a division approach that analyzes both the application data model and the data flow.

Since the migration to MSA varies from project to project, many authors published case studies concerning certain business domains. Belalie et al. [6] reported their experience with migrating to a cloud-native environment. They emphasize the continuous delivery and importance of service contracts. The implementation of a service can evolve; however service contract should remain the same across all implementation versions.

Gouioux and Tamzalit [17] published a case study of migrating large-scale systems to MSA. They also highlighted the importance of a proper continuous delivery pipeline for its significant reduction in deployment costs. This allows for higher optimal microservice granularity. Regarding MSA integration, the authors argued in favor of lightweight passive choreography over orchestration solutions like the Enterprise Service Bus [8], which can be too heavy for MSA. For the former, they report higher reuse of components as well as significantly decreased response time.

In 2018, Mazzara et al. [11] presented an extensive case study on migrating a bank system. Many of their motivations were common: the system had too many functionalities, the coupling between components was too high, it was hard to understand, and the deployment was complicated due to extensive testing. They have migrated to MSA running at Docker Swarm and introduced choreography based on the messaging system RabbitMQ [19].

Furda et al. [15] sees microservice migration as a promising technique of modernizing monoliths and elaborate on three challenges: multitenancy, stateful, and data consistency.

However, these studies failed to explore motivations related to breaking into new problem domains. The studies generally followed the widely accepted use case for MSA of diving a monolithic application into microservices. We wished to show, through the following case study, that motivation can be found in new problem domains as there some problems which require a cloud-based solution.

4. TARGET DOMAIN : CODE CLONES

Code duplication is when one piece of code mirrors, or closely resembles another. It can happen from copy and pasting portions of code, duplicating the structure of code, or copying components within an application. Code duplication makes codebases harder to maintain by both increasing complexity and size of existing code. When bugs are found in one portion of the codebase, any duplicated regions must also be updated. In modern distributed applications, usually with multiple development teams, this problem can grow exponentially more difficult. Traditionally the problem of code-clone detection has looked at the problem of *intra*-application, or code the is copied between source-code of a single application. However, with the rise of online code-sharing services such as StackOverflow and GitHub, it has never been easier to view or copy the code from other resources. This has caused the problem of code duplication to become more and more pervasive in modern software development. Simple modifications of code found online can allow a developer to present existing code as one's own work. Utilizing cloud-technologies, we can explore at the lesser researched problem of *inter*-application code-clones, or code copied from outside sources into an application and show a cloud-based solution is necessary for detecting code-clones from outside sources.

4.1 Basic Notions on Code Clones

There are four types of identifiable code clones [16, 46]

1. Type 1 clones are exact copies of each other
2. Type 2 clones are also exact copies however they may change some non-structural elements like the names of variables, functions, and classes
3. Type 3 clones are similar in structure but have slight modifications, including adding, deleting, or reordering lines. Clones of this type are common in large codebases where large chunks of code may have been copied and modified to fit the developer's specific need
4. Type 4 clones perform similar tasks but may do so in very different ways, so they are similar semantically

While the first three types of code clones all involve copy and pasted code snippets, type four clones do not. There have been many algorithms and tools developed to find code-clones in repositories. Most of these tools focus on types 1-3, with very few bridging into type 4.

In 2017, an analysis of repositories on GitHub [33] was conducted using a new code-clone detection tool, Sourcerer-CC [41]. This study found that for most applications analyzed, a staggering 80% of their source-code could be found copied elsewhere. Furthermore, the experimenters found that a vast majority of the discovered code-clones were of types 1 and 2. For types 3, they used Sourcerer-CC with a threshold of 80% similarity. This report serves as reasoning for why a consistent integration of code-clone detection is so necessary for the age of open-source and readily available code snippets.

A study [28] on the benefits of incorporating code-clone detection into the development process found that within 6 iterations of the project, a telecommunications application, nearly 1000 code clones, were detected and assessed. This study focused on intra-module only; however, when considering the influence of copying code from outside sources, the possibilities are boundless. Another study [31] found that about 15% of the source code for Linux is an identical copy from elsewhere in the same application. The study [31] also found that 30-50% of copied code sections have a least one different line of code. For extremely large codebases such as Linux, this can result in tens of thousands of lines of identical code. Managing the propagation of bugs through code-clones can be nearly impossible once the code is copied throughout the codebase.

While modern code-clone detection tools excel at finding code-clones within an application, they are unsuited to the task of detecting clones from outside sources without changes to the way they are implemented. These tools are too large and unwieldy for use by developers in modern software development. For code-clone detection tools to be appealing to developers, they must address four main issues we've identified with current implementations of code-clone detection algorithms. We show that the current approach by code-clone detection tools is insufficient for many use cases faced by modern developers. We compare multiple code-clone detection tools and consider the limitations of each in the context of recent software development. We present a solution to the problems discussed by applying a cloud-based architecture to an existing code-clone detection tool and consider two use cases in a case study on our tool that supports the considered features.

4.2 State of the Art

The algorithms for finding code-clones in applications is greatly varied across the different implementations. Some of these algorithms focus on pattern matching [22, 23], some on tokenizing code [41] and some on other methods such as AST or tree matching [1, 2, 7]. Below we go through some of the most important algorithms and tools and discuss the differences and commonalities between them.

Some of the earliest and most seminal work on code-clone detection was in 1992 by Baker [2] and focuses on finding code-clones using line-based comparison using a tool called Dup. This algorithm ignores whitespace and comments for comparison and returns pairs of longest matches of code-clones for visualization. In an extension to the algorithm, Dup can look for parameterized code clones in which the code snippets vary in small details, such as variable names. Using pure matching, Dup found in their experiment, that nearly 24% of the file they analyzed was copied. By using parameterized matching [3, 4], this percentage jumped to 85%. The actual implementation of this matching uses hashing of the lines and then a trie for suffix-tree matching [36]. Like most of the tools mentioned, this algorithm focuses on intra-application code duplication.

In 1995, Baker updated the algorithm [1] to find code duplication in large software systems. This is done using an updated data structure known as a parameterized suffix tree. Using this updated structure, Dup can process over a million lines of code in about 7 minutes. The results from processing the application resulted that 13% of the code was duplicated.

Various unique implementations have been proposed, including one by Johnson in 1993 [23], which uses fingerprinting to find exact matches within an application. The algorithm uses Rabin-Karp fingerprinting [25] and comparison to locate and report on the matches. In 1994, Johnson proposed a new algorithm [22], which replaces distinctive words with a special character resulting in a fingerprinting algorithm for types 1 and 2 clones. Interestingly, a metric-based algorithm was proposed in 1996 by Mayrand [35]. This algorithm used another tool, Datrix [29], to extract metric information for comparison. Later, an algorithm for abstract syntax tree (AST) matching was proposed in 1998 by Baxter [7]. Trees are created and hashed into buckets to reduce the number of comparisons between the trees. An inadequate hash function is purposefully used to account for near-miss code clones. This algorithm allows the efficient finding of types 1-3 code clones in effectively O(N).

Up until now the methods proposed were language-dependent, or would only work for a small number of languages. Those algorithms relied on a language-specific parser to analyze source-code files. In 1999 a method for a language-independent code-clone detection algorithm [12] was proposed. This algorithm works by removing all whitespace within lines and effectively condensing them to language-independent string, and the semantics of the coding language is ignored.

In 2002, one of the most critical advancements in code-clone detection was proposed in the form of CCFinder [24]. CCFinder begins by tokenizing the application's source-code files using lexical rules for the specific language. CCFinder can efficiently tokenize applications consisting of millions of lines of code. The streams of tokenized source-code are transformed using rules specific to the language. These rules, for example, may include parameter replacement and removing unnecessary tokens. CCFinder then utilizes a novel prefix-tree matching algorithm to search common prefixes to identify matches. A divide and conquer approach is used to allow for suffix trees that would be too large to hold in memory. In their experiment on the Java 1.3 JDK [20], it took about 3 minutes for almost 600k lines of code, with nearly 30% discovered as clones. This approach was a novel improvement; however, it still focuses on intra-application code clone detection.

A successor to CCFinder was CP-Miner [31], proposed in 2004. This approach has a similar running time and features to CCFinder; however, it reports a 17-52% improvement on the number of clones it can detect over CCFinder. Furthermore, CP-Miner scales to repositories totaling over 3 million lines of code. The main addition to CP-Miner is the automatic detection of bugs resulting from code clones. This tool was applied to operating systems with impressive results, including finding bugs in both the Linux distribution and Apache web server, which were later patched.

In 2007, Deckard [21] was introduced as a new approach to tree-based code-clone detection. The algorithm used in the Deckard tool would generate vectors from the AST or parse trees generated from the source code. The vectors were then clustered to reduce the number of comparisons needed to be done. The AST or parse trees could be built for a large number of languages. Link CCFinder and CP-Miner, the Deckard tool was applied to large scale repositories, including the Linux kernel, to verify it's scalability.

The tool discussed in the study of GitHub repositories was Sourcerer-CC [41], a token and heuristic-based algorithm that focuses on scaling for large repositories. In addition to using tokens to find code-clones, heuristics and hashes were used to account for a large number of types 1 and 2 code clones. This algorithm serves as the basis for the application we propose.

At the time of writing this paper, we are aware of no code-clone detection tool or algorithm that utilizes either MSA or any kind of cloud-based architecture. All of the aforementioned tools are solely theoretical or local deployment. This limits the tools to the efficiency of the host machine and limits scalability. We propose the first of its kind code-clone detection tool that utilizes cloud-based technologies and is built with deployability and scalability in mind.

4.3 Challenges in the Domain

While the advancements in code-clone detection algorithms and tools in recent years have been expansive and thorough, the tools developed remain difficult for use in everyday software development. We've identified four key issues with the current implementations of code-clone detection tools. Most of the tools we've discussed are quite challenging to set up and use. They often require much configuration on the user's end; sometimes, applications must go as far as to provide virtual machine images with settings already complete so that users can use their tool. Furthermore, these tools are occa-

sionally platform-dependent, making portability and reuse in development a big concern.

We propose that a code-clone detection tool must be (1) cloud-based and deployable. Using a service like Docker [10] to host clone detection tools allows for easy deployment and platform-independence, which makes tools far more comfortable to use in everyday development. The second issue is the lack of integration or standard API among the tools. Each tool is unique, but there are frequent commonalities in features between them. Swapping between tools is an incredibly time-intensive process that is not easily undertaken. We propose that code-clone detection tools must (2) easily integrate into third-party frameworks and continuous integration pipelines. Third, the tools discussed mainly rely on local file storage for storing tokens and heuristics. This is impractical for developers nowadays, especially with cloud-hosted applications and distributed development teams. For ease of use in development, it would be advantageous for the developer to keep a repository of already parsed code to check against to avoid the overhead of repeating preprocessing when checking a new repository for clones. For this reason, we propose that code-clone detection tools (3) adopt a "corpus"-based storage mechanism. This involves keeping a separate repository of already-parsed code snippets to avoid time spent reprocessing code. Lastly, the process of finding the clones is often time-intensive and relatively brute force. The speed for finding clones in a project leaves much to be desired for use in everyday development. We propose one solution to this problem by (4) adoption of an advanced heuristic-based index for selecting and filtering the code snippets into a pool of code-clone candidates, instead of considering the entire pool of code snippets as most modern tools do.

5. CASE STUDY : CORPUS-CC

While there have been many algorithms proposed to solve the problem of detecting code clones, this paper proposes a new application built on a cloud-based architecture that uniformly fits the average software developer's pipeline. Our architecture transforms code-clone detection into a service, able to be easily used and integrated into the software development pipeline. Below we introduce the Corpus-CC code-clone detection tool and walk through two use cases for the tool.

5.1 Introducing Corpus-CC

Corpus-CC[1]is a cloud-based code-clone detection tool with emphasis on integration and speed for the developers. Built on existing developments in code-clone detection, the architecture is designed specifically for ease of use in modern software development. Corpus-CC utilizes a microservice-based pipeline for discovering the code, tokenizing, and code-clone detection. Our decision to use microservices was rooted in scalability and ease of deployment. Each of the microservices can be deployed independently of the operating system of the user, meaning it can be easily used in a development environment where each host configuration may vary.

[1]Corpus-CC is available open-source at https://github.com/cloudhubs/corpus-cc

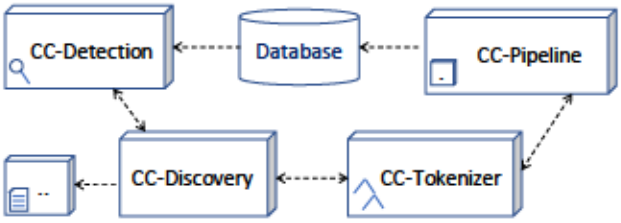

Figure 2: The architecture of the Corpus-CC

Furthermore, multiple instances of each microservice can be deployed for load-balancing. A diagram of the architecture and the interaction between the microservices can be seen in Fig 2.

Corpus-CC is divided into three distinct phases, with each phase corresponding to a specific microservice. The first phase is the discovery phase in which the user specifies a project to be analyzed, and the project's structure and files are scraped. This metadata is then passed to the tokenizing phase, which uses an AST built from each of the code files to extract the code on a method-wise scale. The raw code is parsed into a series of heuristics, which is encapsulated into an object representing each method. These heuristics include the total number of lines, lines of code, and logical lines of code. In addition, the raw code, at varying stages of refinement, is hashed using MD5 [48]. Lastly, the code itself is tokenized. The tokens then undergo a series of transformations which are detailed below -

1. Adding tokens: Tokens are added to add clarity to the structure of the token stream and the conditional structures within the stream
2. Removing unnecessary tokens: Tokens are removed if they are unnecessary to interpreting the structure of the stream
3. Translation of tokens into consistent placeholders: Tokens such as function, class or method names are translated into placeholders

Once the code is tokenized and translated, a count of each type of token and the number of unique tokens is taken. A unique token is any token that underwent translation during the first step. This is for use in the last phase to help reduce the number of comparisons made. This metadata is stored along with the list of tokens. The full information dump for each method is then passed to the third phase for use in code-clone detection.

In the last phase, the metadata for each method is used to query the database of already parsed code using the heuristics and custom queries to drastically decrease the number of comparisons needed to be done. Using the heuristic-based indexing is how we have accomplished a speed increase on the previous methods. The method metadata is compared to the queried results, and clones are added to a centralized report which is presented to the user after all methods have been analyzed. The comparison begins with comparisons of source-code and hashes to account for quick detection of types 1 and 2. If the comparison fails at that stage, then

Table 1: Frequency of Code Clones - StackOverflow

Type 1	Type 2	Type 3	Total
0	506	5,789	6,295

Table 2: Grouping of Code Clones - StackOverflow

Low	Medium	High	Total
4,404	1,387	504	6,295

Jaccard similarity [44] is used to find the similarity between the sets of tokens for each method. The similarity threshold is variable; however, we set ours at 65%. Any pair of methods that exact match or are above the threshold are reported.

To fill the database and create the corpora to compare to, a fourth microservice is used as a pipeline for receiving code snippets. These snippets can come from scraping the web, parsing repositories, or manual upload by the user. This microservice then passes the code onto the previously mentioned tokenizing microservice for tokenization and then loads the metadata into the database. This microservice is designed to be extensible; it is given the various locations from where a developer could want to scrape.

Below we discuss two use cases for our service, StackOverflow and GitHub.

5.2 Use Case I: StackOverflow

With the rise of StackOverflow, and other similar code sharing and help websites, copying code has never been easier. Developers can find code snippets just by searching for keywords related to what they are working on. Issues arise, however, due to the very nature of StackOverflow as a forum for help when *problems* are present in code. The code on websites such as these is unverified, untested, and not always a perfect fit for the project one is working on.

We tested our framework on a corpus loaded with a small amount of code sampled from StackOverflow as a proof of concept. The snippets we chose were from the top results when querying for Java questions. These snippets are a mixture of the top voted answers, the initial broken code, and snippets that StackOverflow independently verified to be the correct answer to the question. In total, we manually scraped about 50 code snippets to use in our corpus. We checked the source-code repository of the Spring Framework [43] against this corpus using our tool.

Even with the small sample size, we found meaningful code clones within the application. The process was quick, not even taking a half a minute to fully analyze the testbed, and most of that time was in the discovery and tokenization process of the application, not in the actual code clone detection. In total, we identified almost 500 methods within our testbed that were code clones with one of the snippets from StackOverflow. In total, code clones accounted for nearly 5% of the complete testbed application.

The snippets were sporadic and not tailored to the testbed beyond the fact that they were the same programming language. As such, we expected a low percentage of matches within the application, purely based around chance matching. We ran another experiment using a separate set of code snippets of similar size and once again found around 5% code-clones within the testbed application. We manually checked the code-clones that were found and were able to verify that they were true code-clones. In both our experiments, all of the code-clones were type 3. This finding is consistent with what we would expect from a developer utilizing code snippets on StackOverflow. As mentioned previously, the code is not always a perfect fit, and so it would be quite rare to find an exact match. An example of a code-clone from this experiment can be found in Listings 1 and 2.

We then moved forward in implementing a web-scraper designed specifically to scrape code snippets off of StackOverflow. We used this scraper to generate a new corpus consisting of 5,000 methods and nearly 50,000 lines of code. We used this corpus along with the Spring Framework testbed to run a new experiment to test the validity of our proof of concept results. The results from testing against the StackOverflow corpus can be seen in Table 1.

To further analyze the clones found, we applied an algorithm for grouping the clones by severity. This algorithm took into account the type of clone, the percentage of the code snippet that was cloned as well as the overall length of the clone in comparison to the others. These clones were grouped and ranked accordingly within each group. We define three groups for code clones - low, medium, and high. A low severity clone is one that was a borderline clone, or a clone with a minimal amount of cloned lines within a code snippet. A medium severity clone is a clone in which a significant portion of the code is cloned. Lastly, a high severity clone is a snippet that is fully cloned or nearly fully cloned. This is beneficial to the developer by reducing a pool of thousands of clones down to just the critical clones that should be addressed immediately. The results from the severity breakdown of the StackOverflow clones can be seen in Table 2.

5.3 Use Case II: GitHub

In addition to code-sharing sites, open-source projects have vastly expanded the number of repositories available for viewing by developers. Much like the concerns with copying code from StackOverflow, the code on GitHub can be unverified, untested, and not a perfect fit. More to the point, however, is that a company with distributed applications may have multiple repositories for an enterprise system. It would be advantageous for a company to create a corpus out of the other repositories for a team to compare their code against, to help avoid duplication across systems. The study [28] previously mentioned has shown the benefits of using code-clone detection within one repository; we now propose a system for detection across distributed repositories.

We loaded a corpus with code taken from the top five repositories on GitHub [13, 37, 39, 40, 42]. In total, these five repositories contained almost 40,000 methods with near 500,000 lines of code to be analyzed. The total time spent to

```
if (isFile()) {
  try {
      return getFile().exists();
  }
  catch (IOException ex) {
    Log logger = LogFactory.getLog(getClass());
    if (logger.isDebugEnabled()) {
        logger.debug("Could not retrieve "
           + getDescription(), ex);
    }
  }
}
```

Listing 1: Code for AbstractResource#exists from the org.springframework.core.io package

```
if (isFile()) {
   try {
      return getFile().exists();
   } catch (IOException ex) {
      System.out.println(ex.getMessage());
   }

}
```

Listing 2: Code found at StackOverflow

Table 3: Frequency of Code Clones - GitHub

Type 1	Type 2	Type 3	Total
1	1,103	9,958	11,062

Table 4: Grouping of Code Clones - GitHub

Low	Medium	High	Total
6,552	4,398	112	11,062

build the corpus, including discovery, tokenizing, heuristic building, and saving was less than 3 minutes. We tested our GitHub corpus against the Spring Framework [43] repository for our testbed application. On average, using the heuristic-based indexing, for each method we tested, we saw an average reduction in the total code-clone candidate pool of around 80% when querying for possible code-clones. This drastically sped up our testing, making it feasible for use in a continuous integration pipeline, or even integration into third-party plugins. Our GitHub corpus is a much larger corpus than the StackOverflow, and so we found a significantly higher number of code-clones. In total, we found that out of the nearly 13,000 methods in our testbed application, 86.7% were code-clones. A breakdown of the types of code clones can be seen in Table 3. We manually checked about 5% of the code-clones to check our threshold metrics were sufficient and were able to verify that the clones we checked were indeed valid code-clones.

These results are consistent with findings from a previous study [33] on code-clones in popular GitHub repositories. That study found that nearly 80% of all files in the GitHub repositories they tested weren't unique and were easily found, in some form, in other repositories. Our consistency with previous findings further verified that our thresholds were correct. The experiment as a whole was a successful demonstration of the nature of using code-clone detection as a service. We showed that it is possible to use a cloud-based application and preloaded repositories of snippets to achieve both the speeds [31] that other tools have reached and the accuracy of previous studies.

We applied the previously mentioned ranking algorithm to the code clones from our experiment with results seen in Table 4.

6. CONCLUSION

Microservice architecture, and on a larger scale, cloud-based architectures are here for the foreseeable future, and it is important to continue to explore and utilize the benefits of these architectures. Even though much research has been done on the process of deconstructing monolithic systems into microservices, not enough research has been done exploring new problem domains using cloud-based solutions. In this paper, we chose to take a look at the problem domain of code-clone detection, one which has existed for decades but is stagnant in innovation. We show that the current implementations of code-clone detection tools are insufficient for the needs of the modern development industry. More specifically, the current implementations lack deployability, are too difficult to integrate, lack efficient storage mechanisms, and are too slow for modern software development. We proposed an upgrade to existing algorithms in the form of Corpus-CC, a cloud-based architecture for integrating code-clone detection as a service for use in the software development pipeline. We verified the validity of use for such a tool in two use cases by running a testbed against common sources of outside code-clones, StackOverflow and GitHub. Even though the code snippet repository for StackOverflow was small, a meaningful number of clones were found in the sample application, and a much larger number of clones were found when comparing against the GitHub corpus.

Creating such a solution would not have been possible without utilizing the benefits of a cloud-based code-clone detection tool. Even though at present, our tool merely wraps an existing code-clone detection solution, we encourage research within this problem domain to explore a transition into cloud-based applications. It is time for the creators of these code-clone detection algorithms to focus on the modern developer and allow their tools to be useful in the new age of software development. Let this paper serve as a challenge to all other tools that they must increase the usability of their tools. As of this moment, our tool, Corpus-CC, is the only tool that addresses any of the problems we've identified.

7. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1854049

8. REFERENCES

[1] B. S. Baker. "On finding duplication and near-duplication in large software systems". In: *Proceedings of 2nd Working Conference on Reverse Engineering*. 1995, pp. 86–95. DOI: 10.1109/WCRE.1995.514697.

[2] B. S. Baker. A program for identifying duplicated code. Computing Science and Statistics, 1992.

[3] B. S. Baker. "A Theory of Parameterized Pattern Matching: Algorithms and Applications". In: *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*. STOC '93. San Diego, California, USA: ACM, 1993, pp. 71–80. ISBN: 0-89791-591-7. DOI: 10.1145/167088.167115. URL: http://doi.acm.org/10.1145/167088.167115.

[4] B. S. Baker. Parameterized Pattern Matching: Algorithms and Applications. Journal of Computer and System Sciences 52.(1): 28 –42, 1996. ISSN: 0022-0000. DOI: https://doi.org/10.1006/jcss.1996.0003. URL: http://www.sciencedirect.com/science/article/pii/S0022000096900033.

[5] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. IEEE Software 33.(3): 42–52, 2016.

[6] A. Balalaie, A. Heydarnoori, and P. Jamshidi. "Migrating to cloud-native architectures using microservices: an experience report". In: *European Conference on Service-Oriented and Cloud Computing*. Springer. 2015, pp. 201–215.

[7] I. D. Baxter et al. "Clone detection using abstract syntax trees". In: *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. 1998, pp. 368–377. DOI: 10.1109/ICSM.1998.738528.

[8] T. Cerny, M. J. Donahoo, and M. Trnka. Contextual Understanding of Microservice Architecture: Current and Future Directions. SIGAPP Appl. Comput. Rev. 17.(4): 29–45, Jan. 2018. ISSN: 1559-6915. DOI: 10.1145/3183628.3183631. URL: http://doi.acm.org/10.1145/3183628.3183631.

[9] P. Di Francesco, P. Lago, and I. Malavolta. "Migrating towards microservice architectures: an industrial survey". In: *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE. 2018, pp. 29–2909.

[10] Docker Inc. *Docker, Enterprise Container Platform*. 2019. URL: https://www.docker.com (visited on 06/10/2019).

[11] N. Dragoni et al. Microservices: Migration of a mission critical system. arXiv preprint arXiv:1704.04173, 2017.

[12] S. Ducasse, M. Rieger, and S. Demeyer. "A language independent approach for detecting duplicated code". In: *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*. 1999, pp. 109–118. DOI: 10.1109/ICSM.1999.792593.

[13] ElasticSearch. *ElasticSearch: a distributed, RESTful search and analytics engine*. 2019. URL: https://www.elastic.co/products/elasticsearch.

[14] K. Finnigan. Enterprise Java Microservices. Manning Publications, 2019. ISBN: 978-1617294242.

[15] A. Furda et al. Migrating enterprise legacy source code to microservices: on multitenancy, statefulness, and data consistency. IEEE Software 35.(3): 63–72, 2017.

[16] P. Gautam and H. Saini. "Various Code Clone Detection Techniques and Tools: A Comprehensive Survey". In: Aug. 2016, pp. 655–667. ISBN: 978-981-10-3432-9. DOI: 10.1007/978-981-10-3433-6_79.

[17] J.-P. Gouigoux and D. Tamzalit. "From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE. 2017, pp. 62–65.

[18] S. Ihde. *From a Monolith to Microservices1 REST: The Evolution of LinkedIn's ServiceArchitecture*. 2019. URL: https://www.infoq.com/presentations/linkedin-microservices-urn.

[19] P. S. Inc. *Rabitt MQ*. 2019. URL: https://www.rabbitmq.com/ (visited on 06/10/2019).

[20] *Java JDK 1.3*. https://docs.oracle.com/javase/1.3/docs/api/. 2000. (Visited on 09/10/2019).

[21] L. Jiang et al. "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones". In: *29th International Conference on Software Engineering (ICSE'07)*. 2007, pp. 96–105. DOI: 10.1109/ICSE.2007.30.

[22] Johnson. "Substring matching for clone detection and change tracking". In: *Proceedings 1994 International Conference on Software Maintenance*. 1994, pp. 120–126. DOI: 10.1109/ICSM.1994.336783.

[23] J. H. Johnson. "Identifying Redundancy in Source Code Using Fingerprints". In: *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*. CASCON '93. Toronto, Ontario, Canada: IBM Press, 1993, pp. 171–183. URL: http://dl.acm.org/citation.cfm?id=962289.962305.

[24] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering 28.(7): 654–670, 2002. ISSN: 2326-3881. DOI: 10.1109/TSE.2002.1019480.

[25] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development 31.(2): 249–260, 1987. DOI: 10.1147/rd.312.0249.

[26] H. Knoche and W. Hasselbring. Using microservices for legacy software modernization. IEEE Software 35.(3): 44–49, 2018.

[27] S. Kramer. *The Biggest ThingAmazon Got Right: The Platform*. 2019. URL: https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/.

[28] B. Lague et al. "Assessing the benefits of incorporating function clone detection in a development process". In: *1997 Proceedings International Conference on Software Maintenance*. 1997, pp. 314–321. DOI: 10.1109/ICSM.1997.624264.

[29] S. Lapierre, B. Laguë, and C. Leduc. Datrix&Trade; Source Code Model and Its Interchange Format: Lessons Learned and Considerations for Future Work. SIGSOFT Softw. Eng. Notes 26.(1): 53–56, Jan. 2001. ISSN: 0163-5948. DOI: 10.1145/505894.505907. URL: http://doi.acm.org/10.1145/505894.505907.

[30] A. Levcovitz, R. Terra, and M. T. Valente. Towards a technique for extracting microservices from monolithic enterprise systems. arXiv preprint arXiv:1605.03175, 2016.

[31] Z. Li et al. "CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code". In: *OSDI*. 2004.

[32] Linux Foundation. *Kubernetes Container Orchestration*. 2019. URL: https://kubernetes.io (visited on 06/10/2019).

[33] C. V. Lopes et al. DéJàVu: A Map of Code Duplicates on GitHub. Proc. ACM Program. Lang. 1.(OOPSLA): 84:1–84:28, Oct. 2017. ISSN: 2475-1421. DOI: 10.1145/3133908. URL: http://doi.acm.org/10.1145/3133908.

[34] T. Mauro. *Nginx - Adopting Microservices at Netflix:Lessons for Architectural Design*. 2019. URL: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[35] Mayrand, Leblanc, and Merlo. "Experiment on the automatic detection of function clones in a software system using metrics". In: *1996 Proceedings of International Conference on Software Maintenance*. 1996, pp. 244–253. DOI: 10.1109/ICSM.1996.565012.

[36] E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. J. ACM 23.(2): 262–272, Apr. 1976. ISSN: 0004-5411. DOI: 10.1145/321941.321946. URL: http://doi.acm.org/10.1145/321941.321946.

[37] *OkHttp*. 2019. URL: https://github.com/square/okhttp (visited on 09/23/2019).

[38] Z. Ren et al. "Migrating Web Applications from Monolithic Structure to Microservices Architecture". In: *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*. ACM. 2018, p. 7.

[39] *Retrofit*. 2019. URL: https://github.com/square/retrofit (visited on 09/23/2019).

[40] *RxJava*. 2019. URL: https://github.com/ReactiveX/RxJava#rxjava-reactive-extensions-for-the-jvm (visited on 09/23/2019).

[41] H. Sajnani et al. "SourcererCC: Scaling Code Clone Detection to Big-code". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: ACM, 2016, pp. 1157–1168. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884877. URL: http://doi.acm.org/10.1145/2884781.2884877.

[42] *Spring Boot*. 2019. URL: https://spring.io/projects/spring-boot (visited on 09/23/2019).

[43] *Spring Framework*. 2019. URL: https://github.com/spring-projects/spring-framework (visited on 09/25/2019).

[44] N. H. Sulaiman and D. Mohamad. "A Jaccard-based similarity measure for soft sets". In: *2012 IEEE Symposium on Humanities, Science and Engineering Research*. 2012, pp. 659–663. DOI: 10.1109/SHUSER.2012.6268901.

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

[46] A. Walker, T. Cerny, and E. Song. Open-Source Tools and Benchmarks for Code-Clone Detection: Past, Present, and Future Trends. SIGAPP Appl. Comput. Rev. 19.(4): 28–39, Jan. 2020. ISSN: 1559-6915. DOI: 10.1145/3381307.3381310. URL: https://doi.org/10.1145/3381307.3381310.

[47] A. Walker et al. "On Automated Role-Based Access Control Assessment in Enterprise Systems". In: *Information Science and Applications*. Ed. by K. J. Kim and H.-Y. Kim. Singapore: Springer Singapore, 2020, pp. 375–385. ISBN: 978-981-15-1465-4.

[48] Z. Yong-Xia and Z. Ge. "MD5 Research". In: *2010 Second International Conference on Multimedia and Information Technology*. Vol. 2. 2010, pp. 271–273. DOI: 10.1109/MMIT.2010.186.

ABOUT THE AUTHORS:

Andrew Walker is a senior computer science undergraduate at Baylor University. His areas of research are verification of distributed systems, static-code analysis and code-clone detection. He is a member of Upsilon Pi Epsilon and ACM.

Tomas Cerny is a Professor of Computer Science at Baylor University. His area of research is software engineering, code analysis, security, aspect-oriented programming, user interface engineering and enterprise application design. He received his Master's, and Ph.D. degrees from the Faculty of Electrical Engineering at the Czech Technical University in Prague, and M.S. degree from Baylor University.