

Maintainability Analysis of Open-Source Serverless Repositories

Matheus Pierre Sforça Breno Bernard Nicolau de França

Relatório Técnico - IC-PFG-24-31

Projeto Final de Graduação

2024 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

The contents of this report are the sole responsibility of the authors.
O conteúdo deste relatório é de única responsabilidade dos autores.

Maintainability Analysis of Open-Source Serverless Repositories

Matheus Pierre Sforça*

Breno Bernard Nicolau de França*

Abstract

Serverless computing is a considerable recent paradigm in application development. This type of application aims to reduce infrastructure and development costs by abstracting the configuration needed to scale and deploy them. Also, as they are only executed when there is some demand, it promises to optimize the cost of running projects. As it is considered a recent paradigm, good practices and its most recommended use cases are still under research. This study examines open-source serverless applications' structure, organization, and code quality by mining software repositories on GitHub. We also compare our findings with related work to see if we identify the same problems found in the existing literature in the area.

In this study, we selected relevant open-source repositories, analyzed their content with SonarQube and internal code metrics, such as the number of functions per file, and consolidated its findings to highlight common characteristics and possible maintainability problems in this paradigm. In this study, we also discuss the limitations of this approach and propose the next steps for future research.

In our analysis, we found out that Serverless projects are indeed smaller in terms of lines of code and usually have great indices of maintainability when checking for Code Smells. The amount of configuration needed also has great advantages by this paradigm, and it was stated a great incentive on writing focused functions with a low number of lines. However, a lack of consensus was identified regarding structuring these projects. In terms of scalability, we stated that as these applications grow in size, problems related to maintainability are more perceptible due to unused code, large functions, and infrastructure constraints that lead to poor coding practices.

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP

1 Introduction

Serverless computing was broadly adopted in 2014 when AWS Lambda was publicly released. This paradigm has been growing stronger, following the growing relevancy of Cloud Computing. Serverless applications promise to make infrastructure management and scalability easier, which are abstracted from the developers by Cloud Providers. Because of that, developers can focus solely on their code without caring much about scaling their applications depending on the demand. Because of those characteristics and the offer of a pay-per-execution model, serverless seems to be a silver bullet in cases where sporadic or unpredictable traffic is expected. That allowed Serverless to be adopted in various industries, including e-commerce and banking, where dynamic scaling and cost-efficiency are crucial [1].

Besides its growing adoption, research regarding this paradigm remains sparse and fragmented, mainly when discussing maintainability aspects of their code. Initial studies have examined the benefits and limitations of Serverless models in performance, scalability, and cost efficiency [2]. A significant knowledge gap can still be noticed in the Serverless context, and the main improvement research areas are code organization, testing strategy, and maintainability where standards are still emerging [3, 4]. Even though some research in this area has already started to address these points, the need for further empirical studies and standards or guidelines for this type of application remains.

In this project, we continue the work initiated by Cunha and de França (2020) [6], in which we re-evaluate the theoretical studies on the topic to understand how research in the area evolved, aiming at mapping common problems identified in Serverless applications. Also, through mining software repositories, we extracted metrics from more than 40,000 Serverless repositories in GitHub and analyzed them in terms of maintainability to identify common issues and challenges.

The metrics were extracted using two distinct approaches: a static code analysis of each relevant file in the repository and an analysis using the SonarQube tool. Metrics from both approaches were consolidated so an analysis could be done over them, leading to relevant insights and a discussion on how Serverless repositories behave and are structured in practice in terms of maintainability.

2 Theoretical Background

This section reviews the important theoretical foundations used during this work.

2.1 Serverless Computing

Serverless computing is a paradigm popularized with the release of AWS Lambda in 2014. This paradigm aims to reduce the developer’s overhead with infrastructure and resource management while optimizing product costs. The idea behind this type of application is that the code is only executed on demand. In other words, it scales automatically depending on its usage. Cloud Providers that offer products with this paradigm are responsible for guaranteeing resource availability and ensuring the scalability of these applications. In

Serverless products, the developer’s role is to provide code in the form of a function that serves as an entry point for the entire logic that needs to be executed.

This paradigm encourages developing an application code in the form of multiple functions that compose each other, and a few responsibilities are expected for each function. Thus, functions tend to be small in number of lines, and not many configuration lines are expected for these products.

2.2 Static Code Analysis and Regular Expressions

Static Code Analysis refers to the examination of the source code of an application without running or debugging it. This analysis checks the project’s source code to identify some structures and elements, such as unused code, poor coding conventions, and duplications.

Regular expressions (or RegEx) express patterns to be searched and matched within a text. In other words, they allow us to locate and extract information matching a given pattern in a text. In static code analysis, regular expressions are used to extract metrics from code, such as the length of function names.

SonarQube[13] is an open-source tool designed for performing static code analysis on software projects. It checks the source code to extract several quality metrics, providing insights and recommendations for potential improvements.

3 Related Work

Recent studies in the context of Serverless computing have already considered some metrics for analyzing the maintainability and software principle violations, such as the size of the serverless functions, coupling among functions, code duplication, and performance-related metrics like execution speed and latency. Next, we review some recent work addressing the theme, analyzing the biggest challenges of serverless applications.

Leitner et al. [7] published a study on which interviews and surveys were performed to examine Serverless development practices. The main extracted challenges from it are:

- Low reusability of code and serverless functions: “However, so far structured reuse of existing functions from this marketplaces does not appear to be a common practice.”
- Functions being implemented as entire services and not properly modularized, violating the Single Responsibility Principle (SRP): “15% of survey respondents implement an entire service or endpoint using just one function”.
- Vendor Lock-In, as the code heavily depends on provider-specific libraries, tools and structures: “Besides lack of tooling and difficulties of integration testing, vendor lock-in is also named as a pressing issue.”. This text highlights a lack of agreement among different providers when discussing serverless products.

In the study from Adzic and Chatley [8] conducted a case study on which similar challenges, mainly vendor lock-in, were observed: “Vendor lock-in Although very little executable code depends on the Lambda environment itself (the platform just triggers the

execution of a custom event handler), working in a serverless environment can make application code highly dependent on the entire platform.”.

Eskandani and Salvaneschi [9] performed a mining of Open-Source repositories and observed that challenges like projects consisting of a single function with a huge amount of lines of code indeed occurred: “While there are several applications with less than 200 LoC and more than four functions, most have over 800 LoC and only a single function.”. This point, in particular, violates Serverless fundamentals, which enhances the idea of building projects consisting of a few lines and with simple logic, complicating the project in terms of maintainability. Some possible causes for this are using the serverless function in contexts where it’s inappropriate or not adapting the code as desired to this paradigm.

In Yu et al. [10], the study analyzed the performance of Serverless functions in multiple experiments. It was demonstrated that breaking a Serverless function into multiple ones and adjusting the code in a way that one Serverless function called the other improved execution speed and reduced the costs: “Decoupling the parallelizable part in an application might help boost the overall performance with parallel-executed serverless functions”, while larger functions suffered from longer startup latency: “Functions with larger code sizes suffer from longer startup latency due to larger data transmission and package import overhead.” This way, adjusting each Serverless function to be simpler (following, at most, the Single Responsibility Principle) provides huge benefits for Serverless projects. In that way, developing serverless code in a single function isn’t the optimal way of working with this paradigm, as improvements and modularization provide reduced costs and higher performance.

In contrast to these studies, Racicot et al. [11] found evidence that projects using Serverless paradigms indeed benefited from low module coupling and have great-designed functions, adherent to software design principles: “software that use serverless technologies have a low module coupling, their components are well balanced and are independent”. However, this study was limited to a small codebase, which might explain why its findings differ from the other studies.

Finally, Wen et al. [12] analyzed Stack Overflow discussions to identify the developer’s main challenges when using Serverless. They stated that most of the questions were regarding the adaptation to the paradigm, including knowledge gaps and platform-specific restrictions.

That way, considering all of the studies, we can say that the three challenges that appeared the most were:

1. Vendor Lock-In, limiting the portability of Serverless projects.
2. Violations of the Single Responsibility Principle (SRP), on which Serverless projects consist of a single function with complex logic.
3. Limited Reuse, which increases the redundancy of some code in the Serverless context and, thus, increases code duplication.

In the following sections, we will obtain metrics from open-source Serverless repositories to check if the challenges and problems identified match the ones observed in the literature.

4 Research Method

We only extracted repositories from GitHub, mainly due to the large number of open-source repositories available and the public API it offers, which allowed us to do an automated extraction. Even though the API supported our needs, it had some limitations, considering the number of repositories of interest and rate limiting our algorithm. To avoid that, we applied a set of constraints that allowed our *script* to fetch from the GitHub API in adequate time intervals, avoiding flooding it with a bunch of HTTP Requests. A negative impact is that the extraction requires an extended period. Also, we only fetched public repositories, and no analysis was conducted in private repositories. The extracted repositories were selected based on the following criteria:

1. Open-Source serverless applications.
2. Repositories matching supported programming languages: Python and JavaScript.
3. Repositories that were not *plugins* or *frameworks* supporting serverless development.
4. Repositories created after the first Serverless computing product was publicly released.

Running the extraction on October 17th, 2024, a total of 42,230 repositories¹ were fetched. We performed a sample check to validate if our selection criteria were not fetching irrelevant repositories for us. For this, we randomly sampled 50 repositories² and manually analyzed them, checking if they were within our context. With the analysis, it was stated that all the sample repositories met the inclusion criteria, meaning they were Serverless applications. With the repositories included, we followed up with metrics extraction.

4.1 Repositories Inclusion Criteria

Next, we cover how each inclusion criteria was implemented by detailing them.

4.1.1 Open-Source Serverless Applications

To identify Serverless applications, we considered the content of their README files, repository names, or descriptions. In each of these, we scanned for the keywords below (case-insensitive), and if any of them were present, we concluded it was a Serverless project.

1. Serverless
2. Function-as-a-Service
3. FaaS

¹https://github.com/MPSforca/unicamp-undergraduate-thesis/blob/main/src/exploration/extracted_repositories.csv

²https://github.com/MPSforca/unicamp-undergraduate-thesis/blob/main/src/exploration/validation_repositories.csv

4.1.2 Repositories Matching Supported Programming Languages

The analysis script used only supported JavaScript and Python projects — this choice aimed to maximize the number of Serverless repositories we could cover. As stated in [5], Python and JavaScript represent over 90% of all serverless applications, so we are confident we could cover most of the available data. Also, both languages are the main ones and are highly supported by cloud providers offering serverless services, such as AWS, Google Cloud, and Azure.

We used GitHub’s repository filter over the primary programming language to find repositories matching these programming languages.

4.1.3 No Plugins or Frameworks Repositories

To exclude plugins and frameworks from our analysis, we discarded the repositories that contained, in their README file, name or description, any of the following keywords (case-insensitive):

1. Plugin
2. Framework

4.1.4 Repositories Created After the Serverless Paradigm

For our analysis, we included only repositories created after November 13, 2014, i.e., when AWS Lambda was publicly launched. This milestone marked the Serverless paradigm as a practical and widely accessible model for application development.

4.2 Analysis Procedure

The analysis procedure was intended to discover how serverless projects are structured in practice, characterize them in terms of that, and find and discuss possible maintainability issues they may present.

To do so, our analysis procedure was based on the following steps:

1. Selection of the Repositories using Github API.
2. Cloning of the repository locally and metrics extraction for them (which is the subject of the next section).
3. Filtering for relevant repositories, trying to avoid personal projects and abandoned repositories (more of this will be described in the next sections).
4. Using Python libraries to analyze the extracted metrics, which will be more detailed later, and draw visualizations for them. The analysis consisted of studying correlation and the frequency of the metrics, and the visualizations were created using bar charts, histograms, boxplots and scatter plots.

All the metrics were extracted and analyzed in the raw form so that we did not filter them in any way. All the original content of the repositories can be accessed, and the relevant information is described in this text.

5 Results

We conducted a series of analyses on the extracted repositories to identify insights related to their characteristics. Every analysis considered only the information for each repository available at the GitHub platform, and the data collection and analysis script can be found in a GitHub repository ³.

5.1 Programming Language

We analyzed the programming language for each extracted repository (Figure 1). This metric was based on Github’s informed primary language for each repository, which is determined based on the programming language with the highest line count on the project files. Also, for both supported programming languages, we didn’t consider other programming languages that compile for them (e.g., Typescript for JavaScript).

From the collected data, JavaScript was the most frequent programming language in Serverless applications.

5.2 Creation Date

The creation date of the repositories was also analyzed (Figure 2). Here, interesting insights can be taken considering the Serverless trend and its usage.

Fast growth appeared in the middle of 2016. Even though AWS Lambda was released publicly in late 2014, it took around two years for the accelerated creation of projects following this paradigm to be noticeable. We can infer that such a delay indicates the time developers need to understand more about the serverless paradigm to evaluate its applicability.

Between 2019 and 2021, a significant increase in the creation of Serverless repositories can be seen. This indicates the peak of popularity of this paradigm and suggests that Serverless computing started getting a “trendy” approach, getting explored for various use cases. During this period, Serverless was implemented in different contexts, potentially including scenarios that were not a proper fit. Also, this fast adoption can reflect the growth of Cloud adoption.

After 2021, we see a gradual decline and stabilization in the number of repositories created after the peak of the popularity of Serverless applications. This indicates the “trendy” approach over Serverless may have ended, possibly indicating that developers found challenges or limitations with the paradigm. This lower interest in trying Serverless may indicate that developers have recognized some of the most appropriate use cases for the paradigm and also understood some that weren’t ideal, leading to a more selective adoption than the one seen from 2020 until 2022.

³Replication Package: <https://github.com/MPSforca/unicamp-undergraduate-thesis>

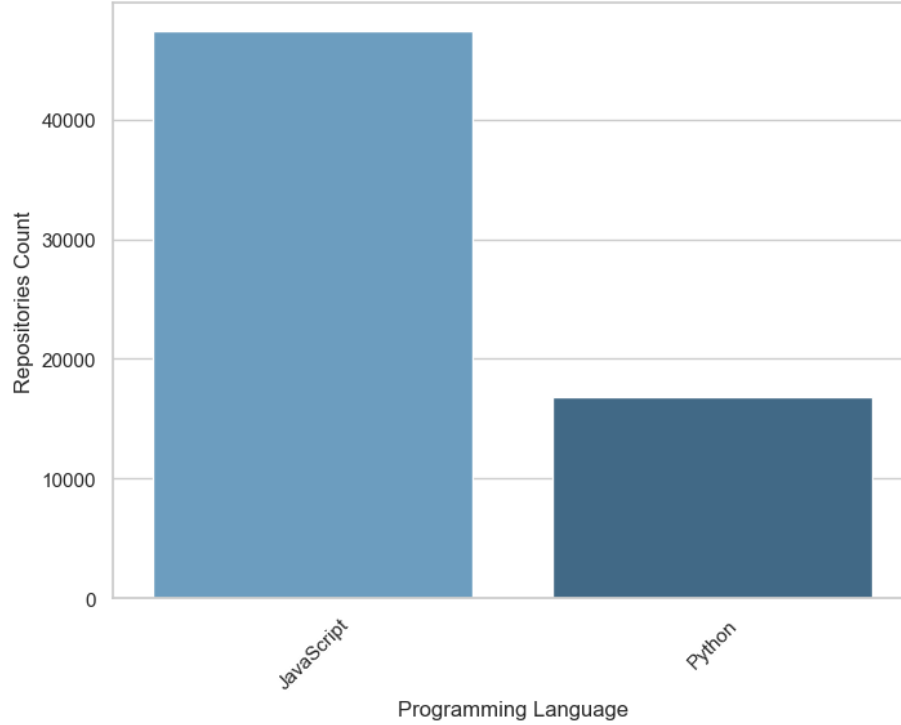


Figure 1: Number of Repositories by Programming Language

5.3 Last Update Date

Examining the last update date of the extracted repositories (Figure 3), many are still being updated, receiving updates at least during 2024. However, a significant number have not been updated since 2023. Inactive repositories may not reflect the latest best practices and should indicate deprecated or abandoned code. Thus, to avoid these scenarios, we will filter for repositories updated within the last 12 months when extracting the metrics.

5.4 Stars Count

The number of stars in a repository indicates its popularity. Analyzing the repositories based on the number of stars (Figure 4), an interesting pattern is that almost all repositories have no stars. Open-source repositories with a zero star count mostly represent small and personal projects, mainly those created simply for experimenting Serverless and to serve as a Proof of Concept. That way, it hardly represents a real application working in a production environment, suggesting they have little to no relevance to industry practices.

When we disregard these zero-star repositories, a balanced distribution may be seen on the remaining, indicating repositories with different levels of popularity. During our metrics extraction, to exclude repositories that may not correctly represent industrial practices (as in the cases mentioned before), we will be excluding repositories with less than ten stars.

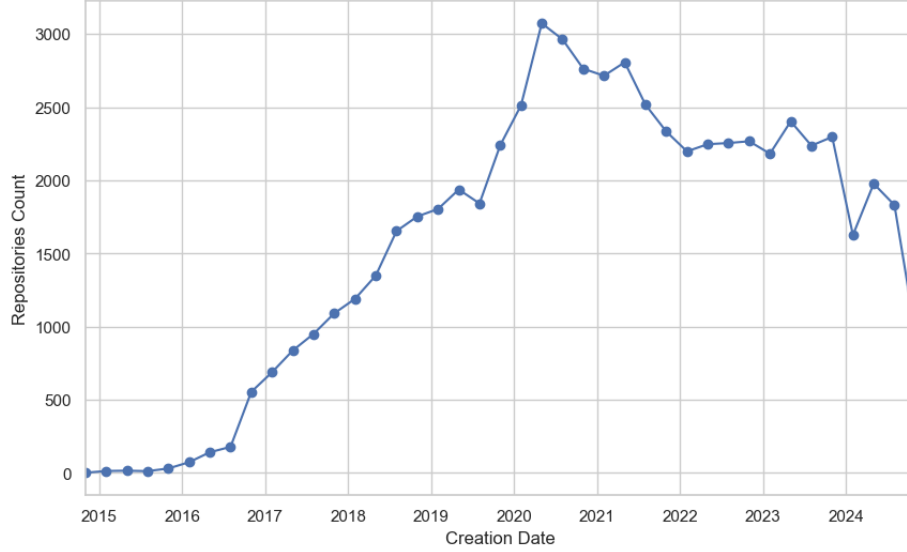


Figure 2: Repositories by Creation Date

6 Metrics Extraction

We extracted metrics from the selected repositories using two different approaches:

1. Static Code Analysis using our scripts.
2. Static Code Analysis using SonarQube.

6.1 Static Code Analysis of Source Codes

The static code analysis of the repository’s source code follows the same principles started by Cunha and de França (2020) [6]. Here, we scanned the repository files for code or configuration files of supported languages.

By code files, we mean a file with one of the following extensions:

- “.py”, for Python files
- “.js”, for JavaScript files

By configuration files, we mean any ending with any of the following extensions:

- “.json”
- “.yaml”
- “.yaml”
- “.ini”

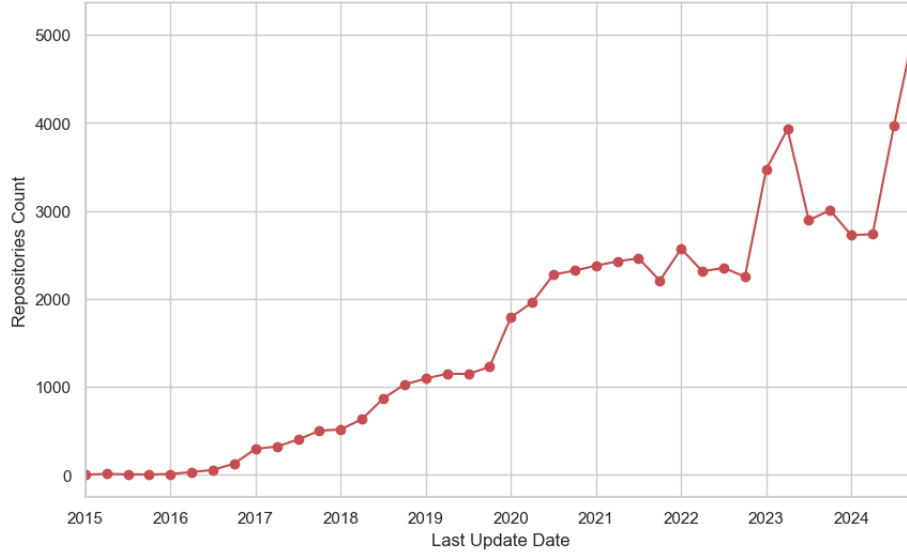


Figure 3: Repositories by Last Update Date

- ".xml"
- ".env"
- ".toml"
- ".properties"
- ".conf"
- ".config"
- ".cfg"
- ".plist"

6.1.1 Metrics

The metrics we extracted during the Static Code Analysis of each repository are condensed in Table 1.

Regular expressions were used to identify and extract useful elements from the code so we could calculate these metrics. Next, we should detail each one and mention that they all continued the work started in [6].

Functions: Identified through the following Regular Expression for each language:

1. Python: `def (.*)\((`
2. JavaScript: `exports\.(.*) = .*`

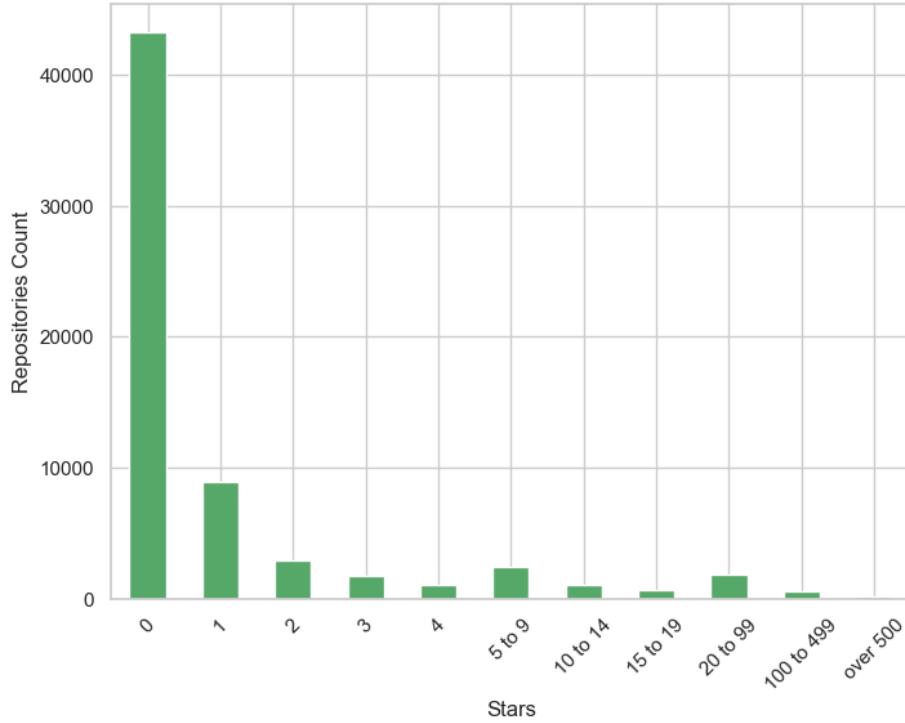


Figure 4: Number of Repositories by Number of Stars (Favorites)

Function Parameters: Identified through the following Regular Expression for each language:

1. Python: `def .*\\((.*)\\).*:`
2. JavaScript: `exports\\..* = .*\\((.*)\\)`

Comments: Identified through the following Regular Expression for each language:

1. Python: `^#.*$`
2. JavaScript: `(\\/\\*.*)|(\\/\\/\\.*)`

Cloud Provider: To detect the Cloud Provider for each repository, we tried to identify the Serverless library used. Each Cloud Provider requires a specific library to be imported into the code.

1. Azure
 - (a) Python: Repositories containing the string “azure.functions” in any of its configuration files or containing the file ”function.json”
 - (b) JavaScript: Repositories containing the string “azure/functions” in any of its configuration files or containing the file ”function.json”.

Metric Group	Metric
Cloud Provider	Cloud Provider
Directories	Number of directories in root Maximum number of code files per folder Average number of code files per folder Minimum number of code files per folder
Files	Total number of files Number of code files Number of config files
Functions	Number of functions Total number of functions Maximum number of functions per code file Average number of functions per code file Minimum number of functions per code file Maximum function name length Average function name length Minimum function name length Maximum number of parameters in functions Average number of parameters in functions Minimum number of parameters in functions
Comments	Maximum number of comments per file Average number of comments per file Minimum number of comments per file
Lines	Maximum number of lines in code files Average number of lines in code files Minimum number of lines in code files Maximum number of lines in configuration files Average number of lines in configuration files Minimum number of lines in configuration files Number of lines of code Number of lines of configuration

Table 1: Metrics extracted from the repositories

2. AWS

- (a) Python: Repositories containing the string “boto3” in any of its configuration files.
- (b) JavaScript: Repositories containing the string “aws-sdk” in any of its configuration files or the function ”exports.handler”.

3. Google Cloud

- (a) Python: Repositories containing the string “google.cloud” in any of its configuration files.

- (b) JavaScript: Repositories containing the string “google-cloud”.

6.2 Static Code Analysis using SonarQube

Besides scanning the files for each repository, we also run the SonarQube tool [13] in the root of each repository to fetch its obtained quality metrics. It offered different metrics from the ones we already had.

6.2.1 Metrics

Code Smells: Sonar classifies Code Smells as “categories of issues in code that may lead to deeper problems over time. They are similar to warning indicators arising from potential bad coding practices.” [14]. Here, considered problems are long functions and language-specific bad practices.

Bugs: Sonar classifies bugs as “a coding mistake that can lead to an error or unexpected behavior at runtime.” [15]. An example of a problem of this type would be a possible division by 0 in the code.

Duplicated Lines: it measures the code’s absolute number of repeated lines. Even though a huge number of duplicated lines may be considered a code smell in the code, this metric was analyzed separately from Code Smells so we could characterize Serverless repositories in terms of it. The idea is to identify insights into how much serverless computing tends to incentivize duplicating lines among the repositories or if its foundation of a small number of lines of code positively impacts it.

Issues: For Sonar, Issues groups all the possible problems (like Code Smells and Bugs) the code may represent [15]. They are classified in the following Severity Levels:

1. Blocker: Bug with a high probability of impacting the application’s behavior in production (such as Memory Leaks).
2. Critical: A bug with a low probability of impacting production or a security flaw.
3. Major: Quality problems that may highly impact developer’s productivity (as unused code).
4. Minor: Quality problems that may slightly impact the developer’s productivity (as long lines of code).
5. Info: Only an observation; it isn’t necessarily a problem that needs some kind of action.

All of these severity level definitions were extracted from the tool documentation [15].

6.3 Extraction Procedure

For all of the repositories that matched our inclusion criterion, we extracted the mentioned metrics from a filtered set of them. The rules considered will be discussed further, and

we did this filter to avoid having our analysis “polluted” with inactive code bases or non-production code used for experimentation projects.

These were the filters applied:

1. Only repositories with a minimum of 10 stars were included to avoid projects for personal usage only.
2. We restricted our selection to repositories that had been updated within the last 12 months to avoid abandoned and inactive code bases.

Given that our initial dataset had 42,230 repositories that matched our search criterion, applying the filters reduced the number to 1,598⁴. On all of the remaining, we proceeded with metrics extraction following the steps below for each:

1. **Clone the Repository:** We cloned the repository from Github locally.
2. **File Scanning and Metrics Extraction:** We scanned each file in the repository, performing the Static Code Analysis of Source Codes described before and extracting their metrics.
3. **SonarQube Analysis:** We ran the SonarQube tool in the project’s root folder. Once completed, we fetched its API to gather code quality metrics calculated by Sonar.

7 Analysis and Discussion

This section presents the results of analyzing all the collected metrics and organizes the discussion on detected issues or behaviors of open-source repositories into sections.

7.1 Could Provider

When we look at the frequency of each cloud provider on the analyzed repositories (Figure 5), AWS is by far the most used in the context of Serverless. Besides being the most used cloud provider currently, the fact that it was the first to release a product for serverless computing publicly may have contributed to this dominance.

Both Azure and Google Cloud Platform have very similar usage.

7.2 Directories

When we look at the count of directories in the root folder of each of the repositories (Figure 6), an interesting pattern can be seen: the great majority of the repositories contain around five directories in the root. It may indicate the existence of more condensed projects in size, which could represent Serverless paradigms being followed in practice.

To better analyze patterns regarding the directories, we should also check the relation among the number of codes files per directory (Figure 7). The histogram shows that the

⁴https://github.com/MPSforca/unicamp-undergraduate-thesis/blob/main/src/exploration/metrified_repos.csv

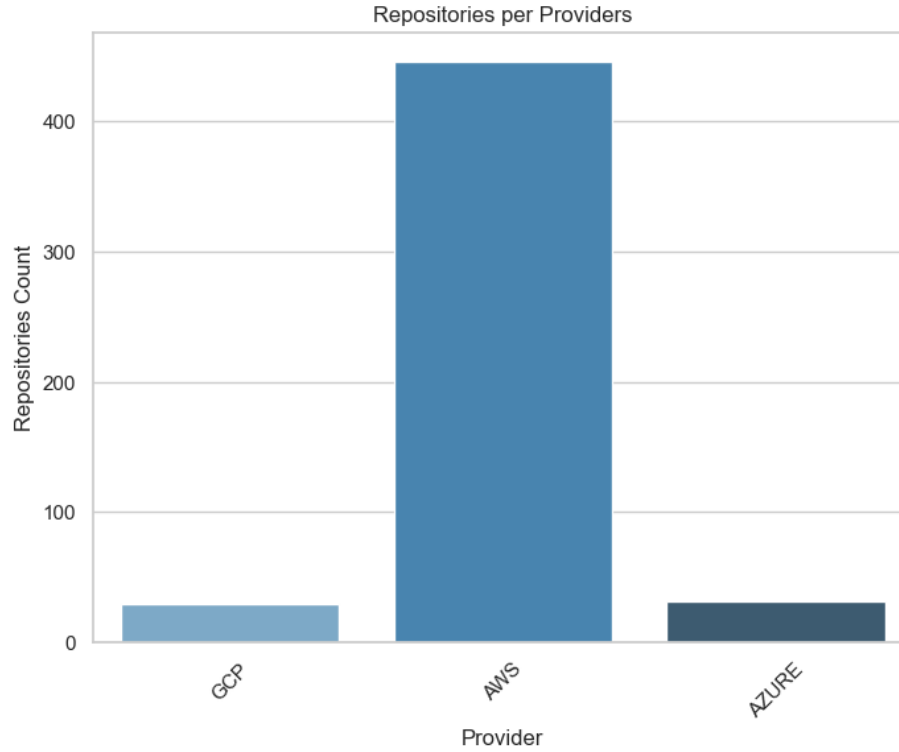


Figure 5: Cloud Providers frequency

minimum number of code files per directory is around three, and the average also has more frequency in lower numbers.

However, when we consider the maximum amount of code files per directory, we can see that its frequency decreases way slower than the minimum amount of code files. Even though this metric remains considerably low, this number may represent a behavior in which code can be organized in an unstructured way, where code for most of the functionalities remains grouped in the same directories. In contrast, the others have smaller components, indicating problems with dividing code for Serverless components.

7.3 Files

For the file count, we have two types: code files and configuration files. The expected structure can be noticed when analyzing their count over all the repositories (Figure 8: overall, repositories have more code than configuration files).

Most of the analyzed repositories maintain up to 30 codes files, which can be an elevated amount in the Serverless context. Still, a deeper analysis regarding this point will be done when analyzing the repository Functions. Regarding the number of configuration files, most of them keep a maximum count of 10, which seems to indicate that serverless projects can be characterized by requiring low configuration.

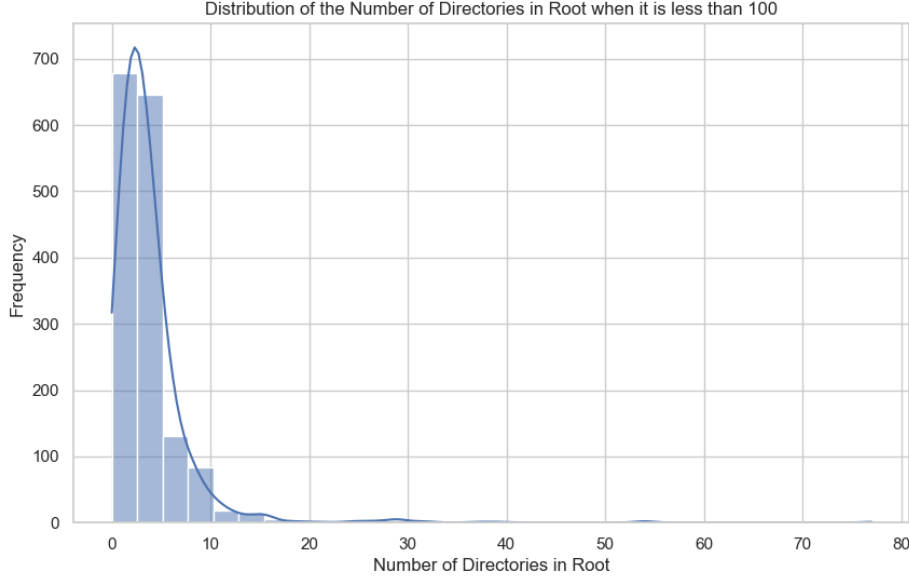


Figure 6: Count of Directories in the Root Folder

Looking at how the density of the number of files, or in other words, the percentage of the number of files by its type over all the files of the repository, behave 9, even more insights can be taken. First, it's noticeable how the number of configuration and codes files differs as the project grows. Most repositories keep a low number of configuration files, and with the code growing in size, that configuration seems to be kept the same. The fast decline in the number of configuration files can state this. When we look only at the number of codes files, we see that the repositories are composed of projects of different natures: from simpler, with fewer files, to more complex ones. That unstable nature indicates no consensus regarding the appropriate size of Serverless projects.

Thus, two patterns regarding these metrics can be stated in the conclusion. The first of all is that **Serverless projects indeed keep a low amount of configuration, which doesn't seem to increase much as the project grows, adhering to what the Serverless paradigm proposes.** Regarding the count of code files, projects vary a lot in terms of that, so there's a lack of consensus regarding how to structure Serverless projects, as developers seem to have not found the optimal size that projects of that nature should have. However, it may also mean that Serverless projects can be implemented for flexible use cases - from smaller to bigger projects. Thus, we will review this element when reviewing the number of functions and coding lines of these repositories.

7.4 Functions

The main aspect of serverless computing is the functions used in the code, which is the element on which the whole code should be based. Because of that, it is essential to analyze the repositories to identify patterns regarding them. Also, an important observation is that

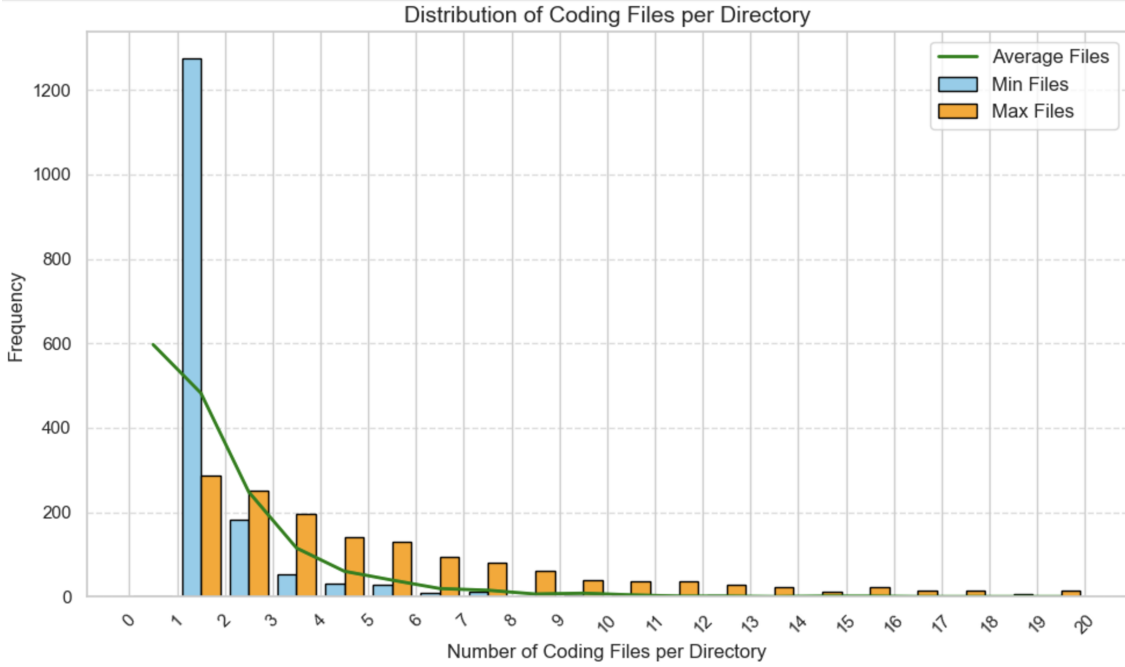


Figure 7: Distribution of codes files per directory

here, we analyze all the project functions (the public, private, and the ones exposed in the Cloud Provider), making no difference among them. The idea is to understand how the projects behave in terms of functions.

Looking at the histogram for the minimum, average, and maximum number of functions per repository (Figure 10), most of the repositories keep their function counts in really low amounts, usually not surpassing 10. Another positive aspect can be stated when looking for the number of functions per code file: files are commonly organized with a small count of functions, even when looking at their biggest code files (the maximum function per code file metric).

Another insight can be taken from here when comparing the histograms of minimum and maximum number of functions: repositories tend to have some files with a low amount of functions, but usually, there are also files with a huge amount of functions. This caught our attention: most of the code is indeed being stored in a single file. That way, **Serverless projects may consist of multiple files on which most have a low number of functions while a single one contains almost all**. Depending on how this is being organized, it can seriously impact maintainability. We should get this discussion back when discussing metrics regarding the project's code files.

Looking now for the name of these functions (figure 11), most functions keep their names within an adequate range of characters (from 10 to 20). This is a great sign that functions can be adequately descriptive in serverless projects, with clear names to indicate what they represent. We can see that the average number of functions is closer to the minimum than the maximum, indicating that some functions with huge name lengths are probably

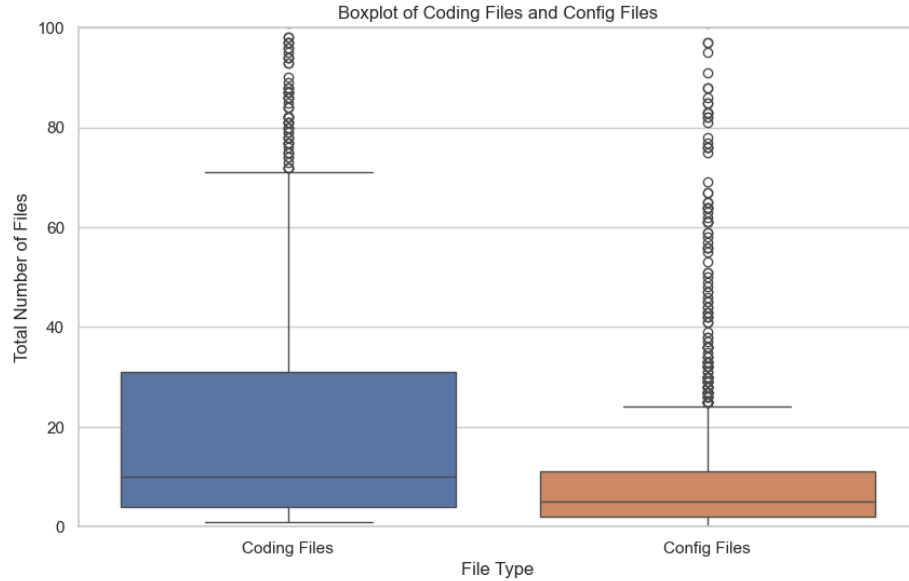


Figure 8: Count of the file types

anomalies in each repository. To understand more of that behavior and why they have such names, an individual analysis could be done in the future, aiming to understand that and maybe check if these functions are the ones that are being exposed in Cloud Providers or not to try to understand more patterns.

Finally, an analysis can be done in the number of parameters of the detected functions (figure 12). Most projects tend to keep a healthy amount - up to three. However, when looking at the maximum number of parameters detected in the code, another alert can be tracked: **with great frequency, there are functions with a lot of parameters in the Serverless projects, which are an indicative of bad coding practice.** In most cases, however, this number is not significantly above average. In the future, an analysis comparing the non-Serverless repositories could be done to understand whether this violation is specific to Serverless applications. In any case, it is still a violation of good coding practices detected in Serverless projects.

7.5 Comments

Comments can be in the code for multiple reasons, signaling things that still must be done, documenting code, or even explaining poorly written code that's hard to understand. When we check the percentage of how comments compose the code files (Figure 13), we check that comments are usually no more than 10% of the total files, which is a healthy count. However, we also see some anomalies in which comments represent over 30% of all the coding lines, which may indicate bad practice. Even though there seems to be a low amount of repositories in that case, a unitary analysis could be done on them to understand if they represent specific cases in which the nature of the Serverless paradigm - having small

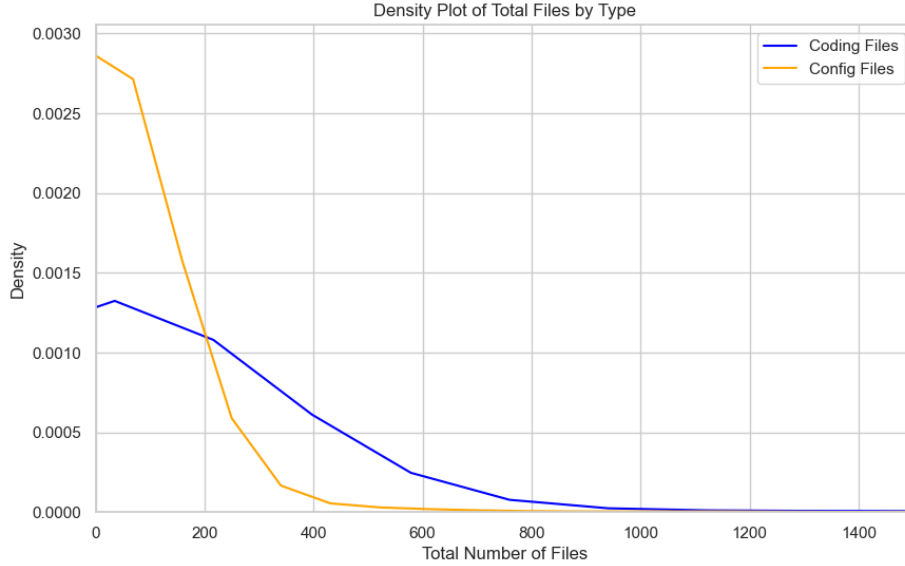


Figure 9: Density of file types per size of repository

and composable functions - may be challenging the readability and maintainability of the code, making it relevant to have so many comments.

7.6 Lines

We analyzed the number of lines of the two types of files we have configuration and coding.

Starting with the number of lines of code of codes files (figure 14), most of the repositories contain no more than 1000 lines of coding. It indicates that these projects properly follow the Serverless paradigm by organizing their project into smaller components. However, we can also state that the peak of the histogram isn't as high as expected - in other words, it demonstrates that projects vary a lot in size (as we had already seen points of that before).

Remembering the behavior of the number of functions in our projects, we can trace another graphic to understand how the number of functions correlates with the number of lines of code (figure 15). Many repositories have a number of functions that grow linearly as the code size increases. In these repositories, we can say that the scalability of Serverless projects tends to occur nicely, as functions keep the same size and, thus, do not get too overloaded with responsibilities. However, many repositories have an opposite pattern - the size of the code grows, but the number of functions does not. This can be stated by the great number of points around the 0 Y-axis, which keeps increasing as the size of repositories increases. This points out an alert, as huge maintainability issues may start to appear here. Functions tend to accumulate many responsibilities, breaking the Single Responsibility Principle.

As a final discussion regarding the number of lines of codes files, the idea that there are still not many patterns or consensus regarding the appropriate size of these repositories got stronger. Besides that, we could state an interesting insight: **as projects grow in**

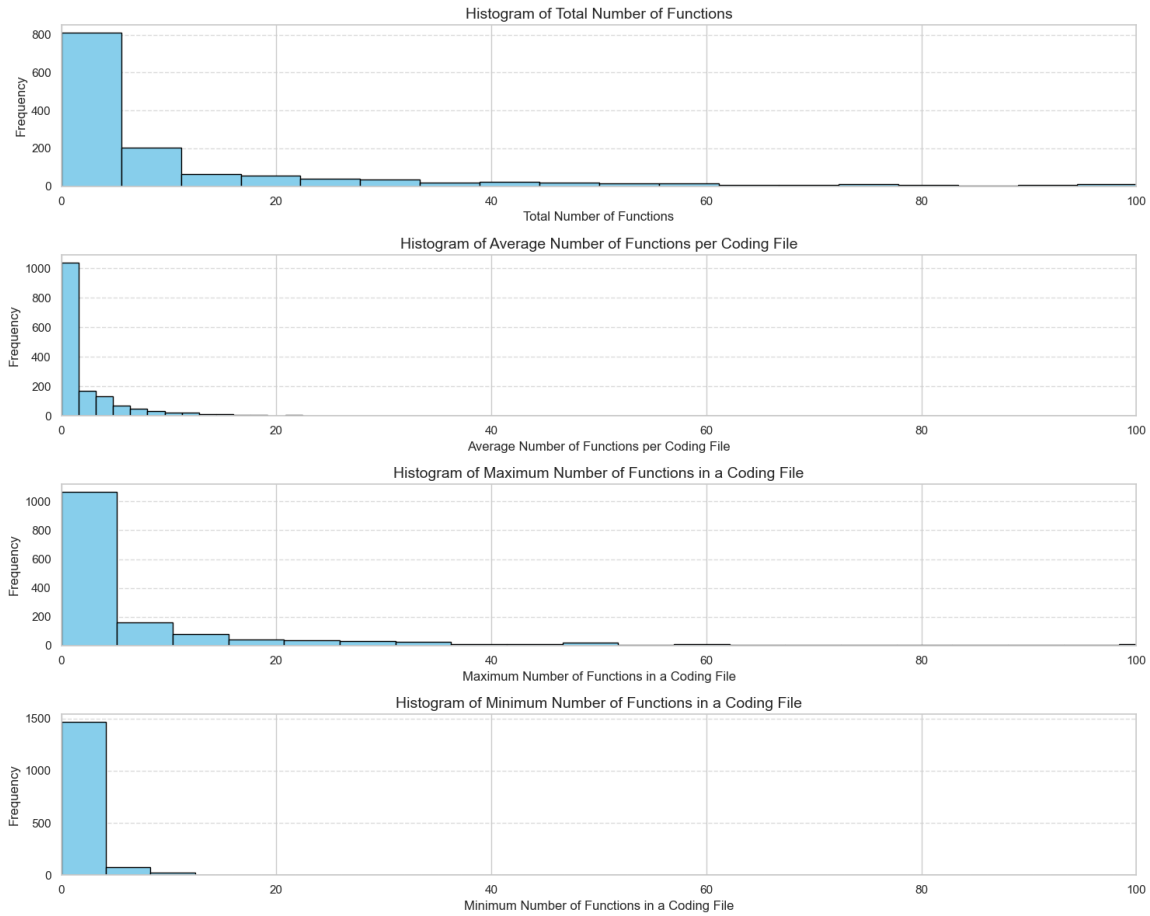


Figure 10: Histograms containing the total, minimum, average, and maximum number of functions

terms of lines of code in Serverless, some tend to keep using a similar amount of functions as initially. Developers may be using the Serverless paradigm erroneously, focusing on its idea of having smaller responsibilities, but it does not seem to be adequate as the project grows. Thus, Serverless may face challenges regarding maintainability for repositories that are too big.

Next, we must make a similar analysis regarding the number of lines of the configuration files (Figure 16). This visualization reinforces our previous statement that, besides the repository size, their configuration tends to be similar, not increasing as the project scales. Thus, a positive benefit of Serverless projects is restated: **as Serverless projects scale in size, we don't see a similar augment for the configuration needed**. This way, a great benefit of Serverless is observed once again: it seems the abstraction of many infrastructure issues by the Cloud Providers and the promise of low configuration needed for those projects are beneficial and can actually be seen in practice.

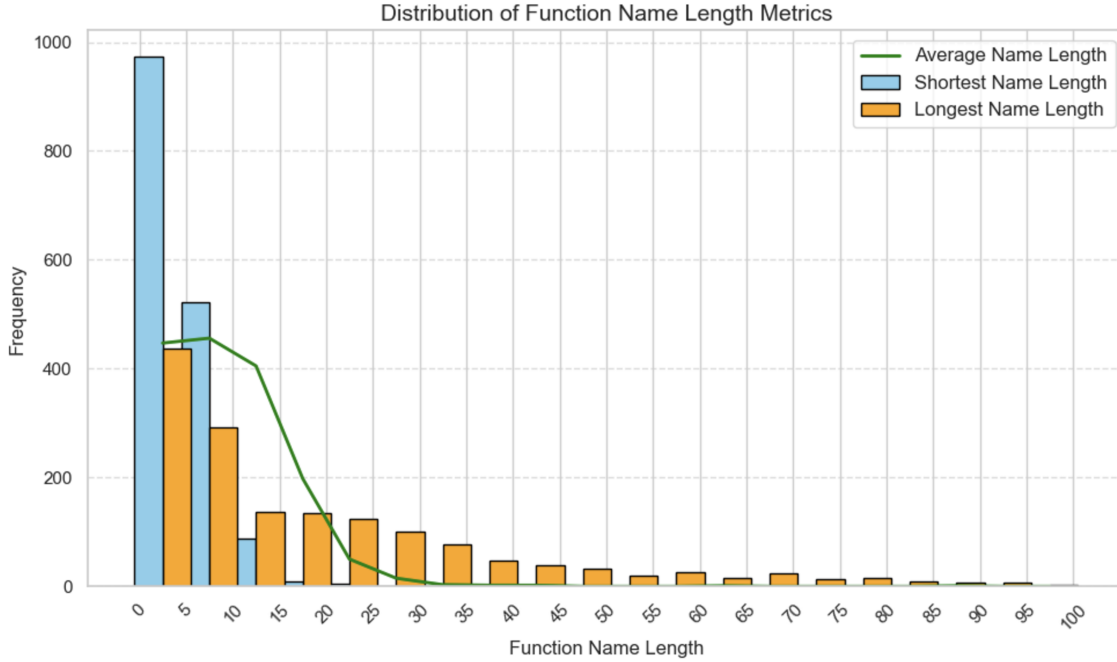


Figure 11: The minimum, maximum, and average length of function names

7.7 SonarQube Code Quality

Now, we can look at the metrics offered by the SonarQube platform for the analyzed repositories.

When looking for the number of code smells (Figure 17) and bugs (Figure 18), similar behavior is observed: the projects have a low amount of these two, with bugs usually being seen as 0. We can consider that Serverless has a meaningful impact on it positively - by encouraging developers to keep their projects small in terms of size and abstracting a huge amount of infrastructure logic, they can focus more on their code in a way that a small amount of code smells are detected. Almost no bugs are found in these projects. Thus, we state a **positive impact in terms of maintainability offered by Serverless paradigm**.

Analyzing especially the amount of duplicated lines, although it is also considered a code smell, we can try to find some characterization among Serverless repositories with this metric (Figure 19). Once again, a great benefit of serverless projects is found - they do not seem to have many duplicated lines, demonstrating a great advantage in the usage of their functions and code, reusing it whenever possible). The smaller size of repositories seems to help developers keep track of something that has already been done before and reuse it whenever possible.

Finally, focusing our analysis on the issues themselves (bugs and code smells). Looking for their severity (Figure 20, on which we counted how many repositories had at least one occurrence of each severity), they tend to be kept among Major and Critical, considering

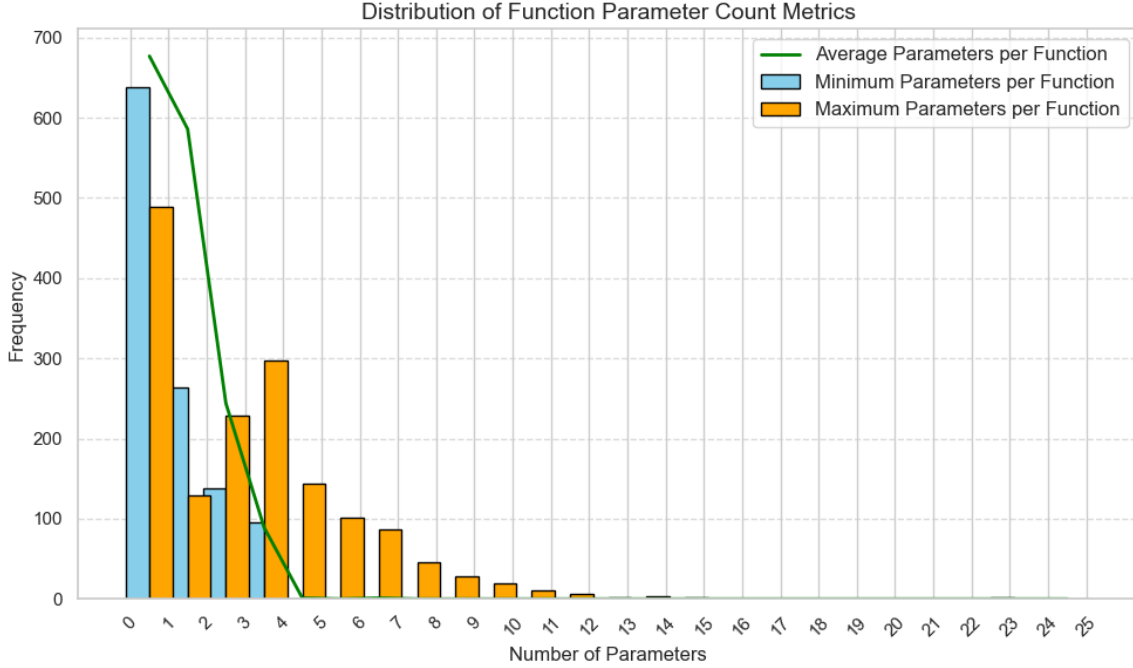


Figure 12: The minimum, average, and maximum count of parameters of the detected functions.

the default SonarQube setup. While critical issues represent problems that may impact production with low chances (being characterized by security flaws, like code susceptible to SQL Injection), major issues represent maintainability issues with high chances of impacting the developer’s productivity (like duplicated or unused code). From this visualization, we can see that most of the issues are usually related to how the code itself is structured, mainly due to the lack of Blocker issues that represent higher severity problems that should impact production, mainly focusing on problems with the communication with other components (like a JDBC connection not closed). Thus, having issues with this classification may be a good signal that in a Serverless context, projects suffer more with how code is written than with performance issues caused by communication with external components.

In conclusion, regarding the severity of the issues, we can conclude the analyzed projects seem to benefit themselves from the Serverless paradigm ideas - as much of its needed code is abstracted by Cloud Providers, the code complexity is greatly reduced, causing in most of the problems detected to be related in how the code is structured itself rather than with problems involving the complexity of enabling or adapting the project to the Cloud.

Each issue is also classified automatically by Sonar in *tags*, which provides a context regarding the “nature” of the problem found, such as unused, for example. For all the issues found in our analysis (Figure 21), most of the problems are related to coding conventions (mainly the ones tagged with “es2015”, “convention,” and “bad practice”), unused code (tag “unused”) or unnecessary complex code (tags “design,” “brain-overload” and “confusing”).

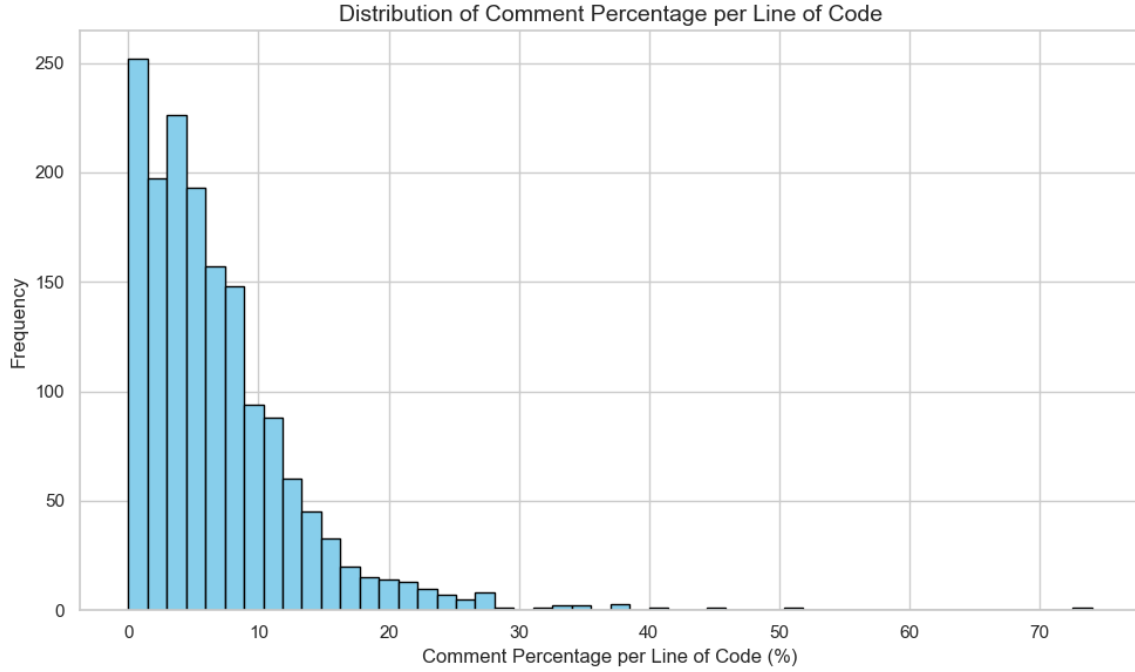


Figure 13: The composition of comments in the number of lines of code

Unused or complex code may be caused by requirements of Cloud Provider libraries for practices such as forcing standards or behaviors when consuming its libraries (like ensuring the configuration of Cloud communication with several parameters, even though many of them are not needed). Coding convention problems may not be related to the Serverless context - they could be problems in any other application.

In Table 2, a complete description of the type of detected issues can be found. All of them were extracted from SonarQube’s rules page [16].

During our analysis, we realized that most of the issues were related only to JavaScript. After an investigation, we discovered the reason: most of our repositories were written in that language, which is why this unbalanced nature was observed. To avoid having the analysis solely for JavaScript code, we split our examination per repository’s main language.

Starting with JavaScript, the ten most common issues were condensed in figure 22, with the descriptions in table 3. Issues S3504, S3776, S125, and S1135 are caused by too big functions that take too much responsibility and should be broken. In contrast to the serverless paradigm, that promotes small and independent functions, in practice we see that this idea is not being properly implemented and followed. When a given application is poorly modularized, these problems may appear, ending up with too big and complex functions that have a bad maintenance rate and with error-prone code. Thus, **poor modularization can be an occurring problem in Serverless, causing functions that are too big or complex**. Now, considering issues S1854 and S6353, they do not seem associated with the Serverless context, as they could appear in any other JavaScript project. Finally issues

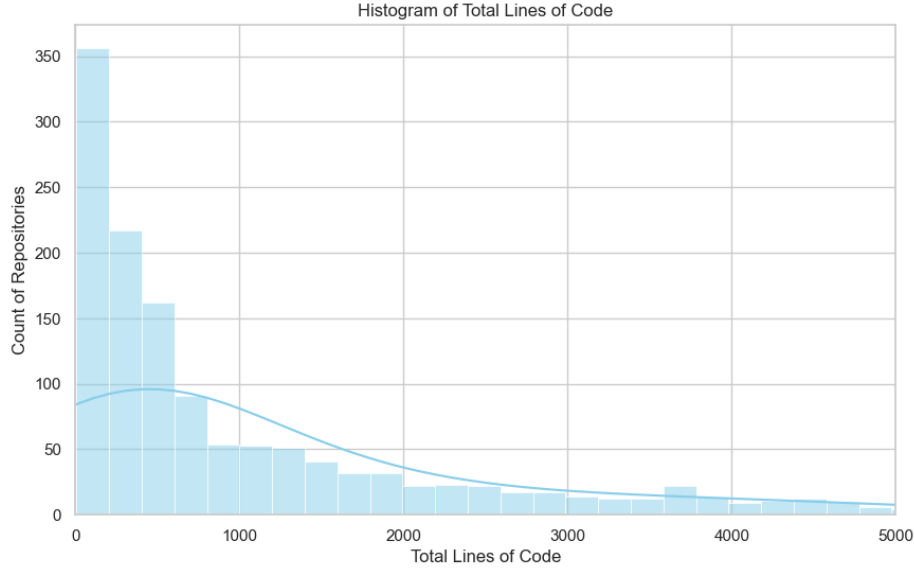


Figure 14: Lines of Code of the Code Files per repository

S6294 and S5254, although not necessarily related to the code maintainability of Serverless projects, are directly related to the paradigm. These issues point out vulnerabilities and problems when configuring Serverless for the Cloud Providers, which can be caused by inadequate usage of Serverless computing, difficulties to understand or find documentation for that, or even limitations when writing code. This way, **these set of elements are causing problems to the Serverless application, compromising its efficiency and even making the code prone to vulnerabilities.**

We should also look for the ten most common issues that were condensed in Figure 23, with the descriptions in table 4. The most common problem is unused code (S1172 and S1481), **which may have an accentuation due to the Serverless paradigm, as some structured requirements may have been imposed by Cloud Providers, such as importing and declaring specific code even when that's not needed.** Problems S1192, S125, S117, S1135, and S5806, even though also related to maintainability, are probably not related to the Serverless context. They are probably caused by the developers not following some conventions in the language, which can be an internal agreement of the project. The rule S3776 points out that **Serverless applications are being composed with excessive complex functions, probably with way more than a single responsibility, which is a violation of the Single Responsibility Principle.** That way, developers may find issues or difficulties when separating their source code into serverless applications, causing poor modularization and structuring of the code. Finally, problem S5754 illustrates that even reducing much of the developer's concerns with infrastructure, **Serverless components may impose some limitations in how they communicate with external services,** probably caused by some restriction of Cloud Providers. That way, Serverless computing may influence code to have vulnerabilities when some use case



Figure 15: The correlation between the number of functions and the number of lines of code of repositories

is still not supported by it (or when a given use case isn’t appropriate for this paradigm).

8 Limitations and Next Steps

There are some limitations in the way we structured our analysis.

The first one is in how we select our repositories. We only used GitHub as the source of content, which implies that we may have excluded interesting repositories stored in other places. Even though GitHub is the most used platform, analyzing the content from different platforms could provide different characteristics for the analysis.

Another limitation relies on our exclusion and inclusion criteria. As we use raw text matching for that (for example, we exclude repositories with the word “framework” in their description), false-positive projects may have mistakenly been excluded and discarded in our tool.

Regarding our Regular Expressions to detect the structures in the source code, we may not have covered all possible cases. This is because we have considered over 10 years of serverless projects. In the meantime, many structures may have been deprecated in the supported programming languages, which our tool could not have considered or covered. In contrast, future work involving this code should review these expressions and patterns to ensure the code is still updated.

Finally, as most of our analysis was automated, some interesting patterns in individual repositories could have been ignored. Because of that, we could review the projects considered anomalies individually to understand whether they are indeed anomalies or just have some specific use case that differs from the others.

Considering these limitations, we should try to address these problems as the next steps.

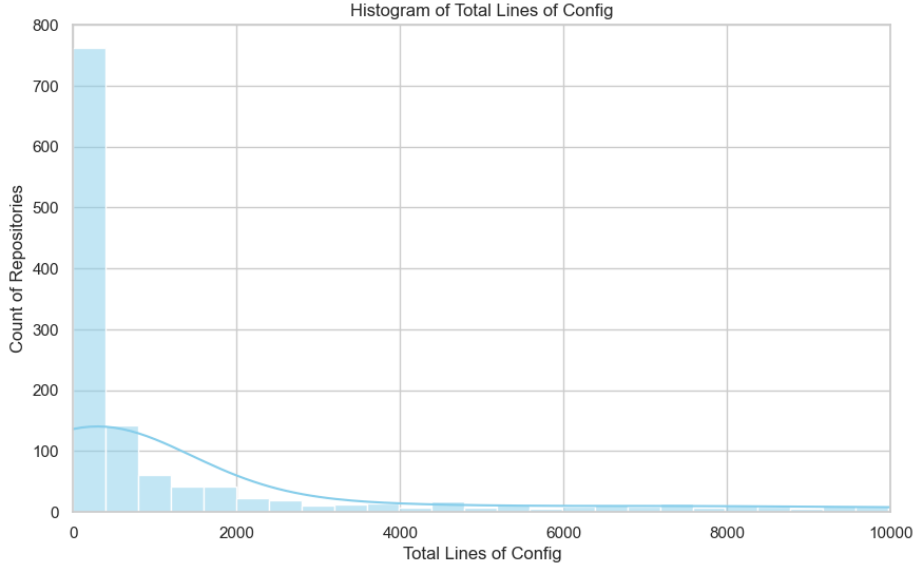


Figure 16: Number of Lines of Configuration Files per repository

This could even improve the highlights we got with our analysis, possibly detecting specific and interesting patterns in some repositories.

First, a manual examination could be done in some selected repositories, mainly in the anomalies, so we can uncover why they behave like that, even providing more insights into how applications are handled and implemented in a Serverless context.

Another potential improvement would be refining the inclusion and exclusion criteria used. Here, we should avoid removing the false-positive repositories simply because they have a specific word in their description.

A final improvement would be improving the relationships among our metrics. We could, for example, do multiple analyses depending on the lines of code of the repositories, so we try to understand if bigger Serverless repositories have different structures or practices compared to smaller ones.

As all of the code used and obtained metrics are open-source, this project was intended to be used to incentivize easy reuse to gain even more insights into how Open-Source Serverless applications behave.

9 Conclusion

In conclusion, several valuable insights involving Open-Source Serverless projects could be obtained even with our discussed limitations, which went beyond maintainability metrics.

In general, we can see that the peak popularity of Serverless was between 2020 and 2022, when most of the open-source repositories were created. After that, it seems to have stabilized. Most of the open-source repositories from that paradigm also seem to be smaller applications, Proof-of-Concepts, and experimental projects, having no real usage.

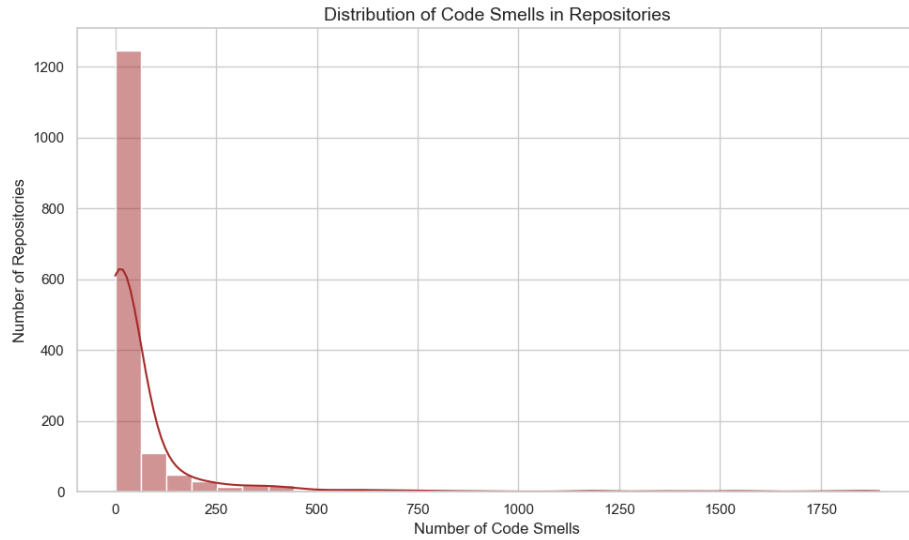


Figure 17: Number of Lines Code Smells per Repository

Analyzing our extractions, we saw that the Serverless paradigm indeed helps with what it proposes: its projects usually have a lower number of Lines of Code and are limited in the number of files. However, it seems that in terms of maintainability, when we need to scale the size of these repositories, problems related to code structure seem to turn more apparent. Unused code is also a relevant issue when working with serverless projects, as stated by the number of issues mentioning that point. Not many Infrastructure problems seem to have been noticed in our analysis, which may point out that the fact that the paradigm aims at abstracting it from the developers seems to have been positive. At the same time, some Issues are related to communication between a Serverless function and another external component, which may point out that Cloud Providers may have some limitations regarding that in this kind of paradigm or that Serverless functions are trying to be used in contexts they are not appropriate for. Regarding the configuration of these applications, it seems that the Cloud Providers are implementing what they promised: almost all Serverless projects have a similar number of configuration lines, indicating that even when growing and scaling the projects, not many extra configurations are needed.

We could confirm that many of the common problems identified in Serverless applications discussed in Related Work also seem to be present in these applications, mainly the difficulty of breaking the code into multiple functions to help its maintainability aspect and be in agreement with the Single Responsibility Principle. Because of that, it's not so uncommon to see Serverless projects consisting of a single function or of a small number of functions composed of numerous amount of coding lines.

In summary, we can say that the serverless paradigm helps the projects in terms of maintainability, as we can measure with our static code analysis metrics. However, some care also needs to be taken to avoid structural problems, mainly when the project grows in size.

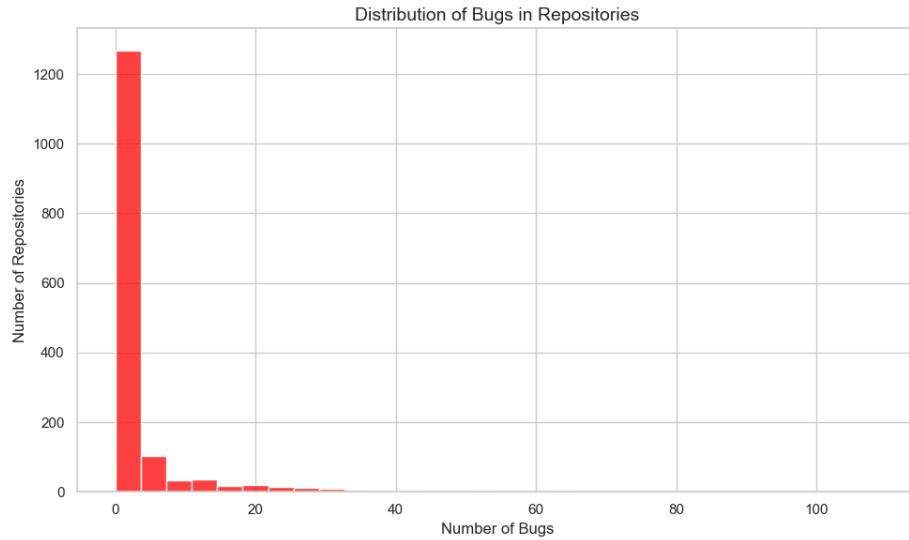


Figure 18: Number of Bugs per Repository

References

- [1] SHAFIEI, Hossein; KHONSARI, Ahmad; MOUSAVI, Payam. Serverless computing: a survey of opportunities, challenges, and applications. *ACM Computing Surveys*, v. 54, n. 11s, p. 1-32, 2022.
- [2] PADYANA, Uday Krishna et al. Server less Architectures in Cloud Computing: Evaluating Benefits and Drawbacks. 2024.
- [3] LAKHAI, Vladyslav; KUZMYCH, Oleh; SENIV, Maksym. An improved method for increasing maintainability in terms of serverless architecture application. In: 2023 IEEE 18th International Conference on Computer Science and Information Technologies (CSIT). IEEE, 2023. p. 1-4.
- [4] MEGHLA, Tamara. Testing Serverless Applications: A systematic Review of Frameworks, Tools & Best Practices. 2023. Dissertação de Mestrado.
- [5] ESKANDANI, Nafise; SALVANESCHI, Guido. The wonderless dataset for serverless computing. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR). IEEE, 2021. p. 565-569.
- [6] CUNHA, Davi Vinícius; DE FRANÇA, Breno Bernard Nicolau. Análise de repositórios de software em projetos baseados na arquitetura Serverless. Universidade Estadual de Campinas, Instituto de Computação, 2020.
- [7] LEITNER, Philipp et al. A mixed-method empirical study of Function-as-a-Service software development in industrial practice. *Journal of Systems and Software*, v. 149, p. 340-359, 2019.

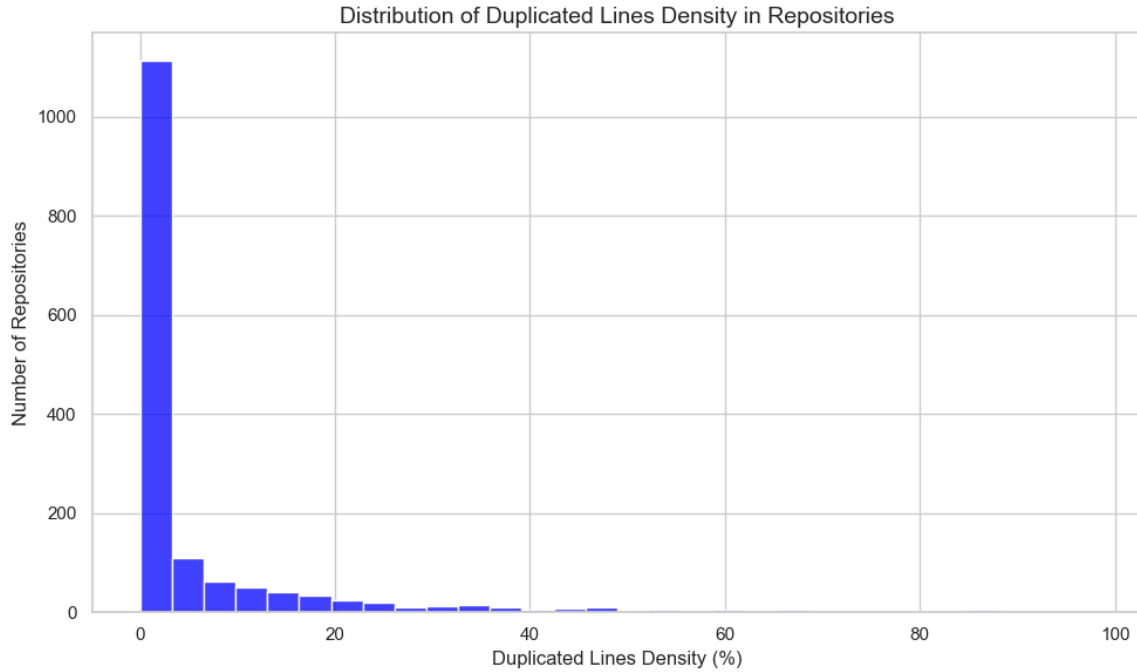


Figure 19: Number of Duplicated Lines per Repository

- [8] ADZIC, Gojko; CHATLEY, Robert. Serverless computing: economic and architectural impact. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering. 2017. p. 884-889.
- [9] ESKANDANI, Nafise; SALVANESCHI, Guido. The uphill journey of FaaS in the open-source community. *Journal of Systems and Software*, v. 198, p. 111589, 2023.
- [10] YU, Tianyi et al. Characterizing serverless platforms with serverlessbench. In: Proceedings of the 11th ACM Symposium on Cloud Computing. 2020. p. 30-44.
- [11] RACICOT, Louis et al. Quality aspects of serverless architecture: an exploratory study on maintainability. 2019.
- [12] WEN, Jinfeng et al. An empirical study on challenges of application development in serverless computing. In: Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering. 2021. p. 416-428.
- [13] SONARQUBE. Documentation. Available at: <https://docs.sonarqube.org/latest/>. Accessed on: Nov. 3, 2024.
- [14] SONARSOURCE. Code Smells. 2024. Available at: <https://www.sonarsource.com/learn/code-smells/>. Accessed on: Nov. 3, 2024.

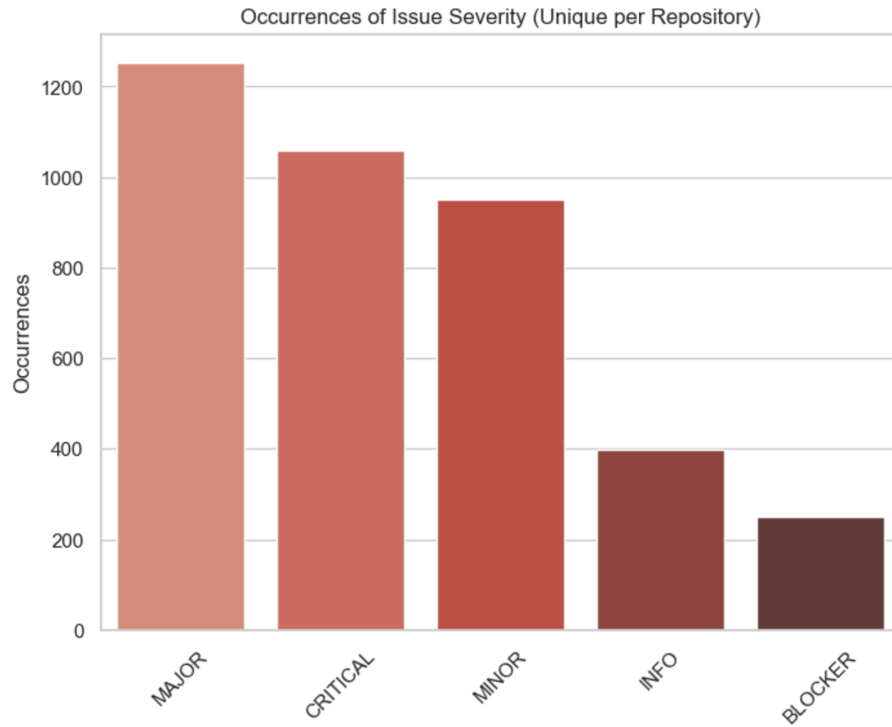


Figure 20: Severity of the detected issues

- [15] SONARSOURCE. Issues. SonarQube Documentation, 2024. Available at: <https://docs.sonarsource.com/sonarqube/9.9/user-guide/issues>. Accessed: Nov. 3, 2024.
- [16] SONARSOURCE. Rules SonarSource. Available at: <https://rules.sonarsource.com/>. Accessed on: December 1, 2024.

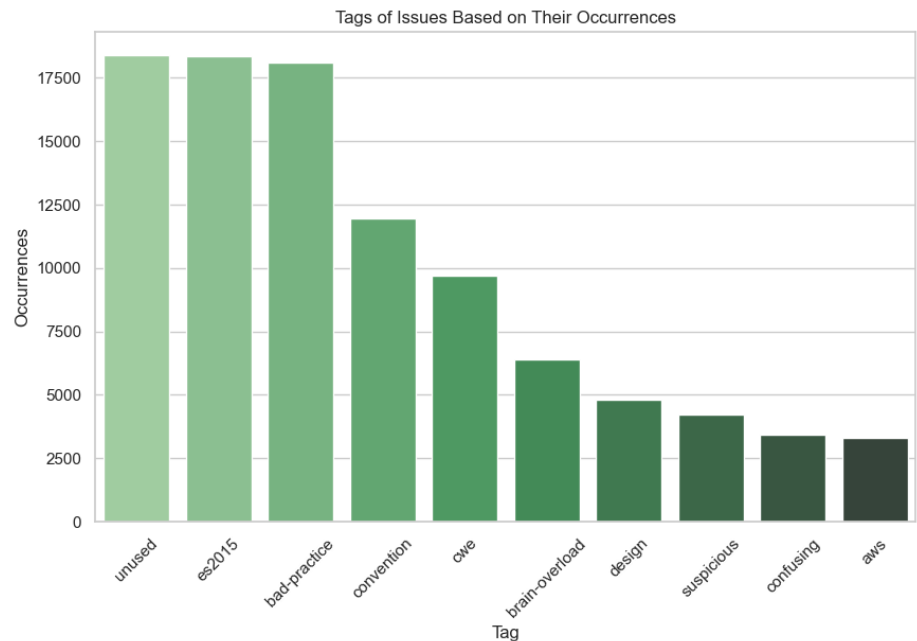


Figure 21: Tags of the detected issues

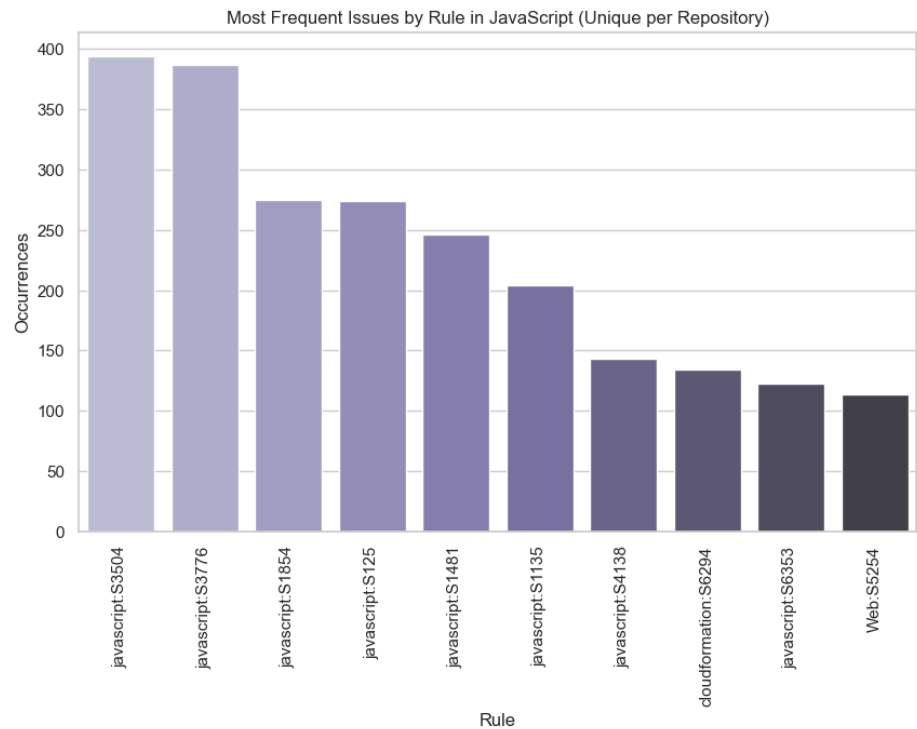


Figure 22: Ten most common JavaScript issues in the repositories.

Category	Description
Unused	Refers to code or variables that are not used. This type of code can be removed to improve readability and reduce project complexity.
ECMAScript	Identifies practices and features specific to ECMAScript 2015 (ES6). Helps ensure compatibility and adherence to modern JavaScript standards.
Bad Practices	Markers for practices that do not follow recommended conventions or that may generate maintenance and quality issues. Encourages adherence to best practices.
Convention	Checks adherence to coding conventions, such as style, formatting, and naming. Promotes standardization across codebases.
CWE	Refers to Common Weakness Enumeration to help identify security vulnerabilities. Facilitates robust and secure software development.
Complexity	Indicates overly complex parts of code that may be difficult to understand or maintain. Suggests simplifications to make code easier to manage.
Design	Evaluates design patterns and principles, such as separation of concerns and modularity. Promotes good engineering practices.
Suspicious	Flags code that appears to contain bugs or unexpected behavior. Requires review to ensure correct functionality.
Confusing	Identifies elements of code that are difficult to understand. Suggests changes to improve clarity and readability.
AWS Practices	Relates to the use of specific AWS practices and configurations. Helps verify secure and efficient cloud configurations.

Table 2: Issues Tags and Descriptions

Rule Code	Description
S3504	<i>Control flow should not be too deeply nested</i> - The rule warns against excessively nested control flows, which make the code difficult to read and maintain
S3776	<i>Cyclomatic complexity should not be too high</i> - The rule checks the cyclomatic complexity of functions, suggesting that functions with high complexity be split to improve readability and testability.
S1854	<i>Variables should not be reassigned</i> -Discourages variable reassignment, promoting the use of <code>const</code> or <code>let</code> to avoid unexpected behavior and make code more predictable.
S125	<i>Methods should not be too long</i> - It recommends that methods be short and have a single responsibility, facilitating reading, maintenance and testing.
S1481	<i>Empty catch blocks should not be used</i> - The rule warns against the use of empty <code>catch</code> blocks, which can suppress errors without handling them, impairing fault detection.
S1135	<i>If-else chains should not be too complex</i> - The rule suggests simplifying very complex <code>if-else</code> chains, improving readability and avoiding errors.
S4138	<i>Duplicated code should be avoided</i> - Identifies and discourages code duplication, suggesting refactoring to reduce redundancies and ease maintenance.
S6294	<i>Lambda functions should have clear timeouts</i> - For Lambda functions on AWS, it is necessary to define clear execution times to prevent them from running indefinitely (using the default timeout) and wasting resources.
S6353	<i>Comparison with literals should not be done</i> - The rule recommends avoiding direct comparisons with literals (e.g., <code>if (x == 0)</code>), to improve readability and avoid unexpected type corrections.
S5254	<i>External resources should be served securely</i> - Ensures that external resources (such as libraries or APIs) are served via HTTPS, protecting against data interception attacks.

Table 3: Rules and Descriptions for JavaScript repositories. Descriptions extracted from [16]

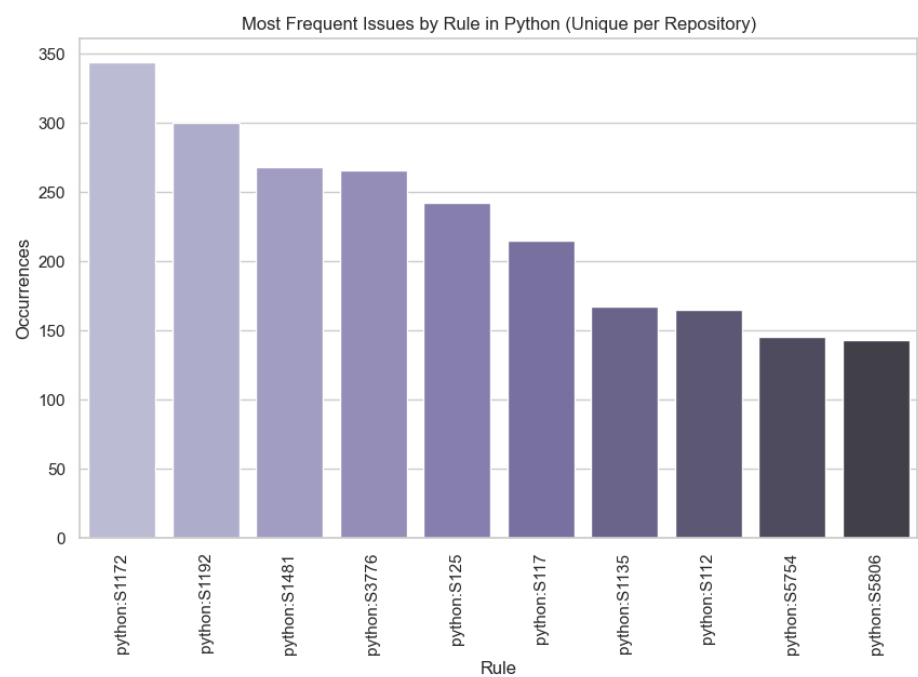


Figure 23: Ten most common Python issues in the repositories

Rule Code	Description
S1172	Identifies unused method parameters in the method body, which may indicate dead code to be removed to improve readability.
S1192	Recommends avoiding duplicating literal strings in your code, as this can lead to maintenance errors if they are not changed consistently.
S1481	Detects unused local variables. These can be eliminated to reduce clutter and improve code clarity.
S3776	Measures the cognitive complexity of methods, suggesting refactoring for complex methods to increase readability and maintainability.
S125	Alert for unnecessary commented code, which should be removed to improve clarity and avoid dead code.
S117	Identifies variable or constant names that do not follow naming conventions, improving code consistency and readability.
S1135	Detects the presence of TODO comments that indicate pending work, recommending that they be resolved or tracked appropriately.
S112	Flags the use of generic exceptions (<i>Exception</i> or <i>Throwable</i>), suggesting the use of more specific exceptions to provide more context and clarity.
S5754	Detects the use of cookies without appropriate security settings, such as <i>HttpOnly</i> or <i>Secure</i> , which may expose the system to vulnerabilities.
S5806	Warns about the use of inefficient regular expressions or those vulnerable to denial of service attacks (<i>ReDoS</i>), recommending a review to improve security.

Table 4: Rules and Descriptions for Python repositories. Descriptions extracted from [16]