# KIREPRO1PE

# Compression of Programs and the Similarity Distance

Jonas Nim Røssum <jglr@itu.dk>

**Supervisor:** Mircea Lungu <mlun@itu.dk>

Lines of code changed (LoCC) is one of the most widely used distance metrics in the software industry, but it suffers from several key drawbacks that can mislead when analyzing developer effort and project activity. In this paper, we have explored the shortcomings of LoCC and propose an alternative metric: Compression Distance, $\mathrm{CD}(x, y) = |C(x)| - |C(x \cup y)|$ derived from Normalized Compression Distance (NCD) and lossless compression algorithms with large search windows. The CD metric has been presented as a complementary metric for information distance to traditional methods like LoCC. The results suggest that CD provides a more nuanced and robust measure of software evolution, to be used as a complementary metric to proven metrics like LoCC, while being aware of the biases in the metric.

May 15, 2025

*They took away the old timbers from time to time, and put new and sound ones in their places, so that the vessel became a standing illustration for the philosophers in the mooted question of growth, some declaring that it* ***remained the same, others that it was not the same vessel.***

— Plutarch, Life of Theseus 23.1

# Contents

# 1. Introduction

Software engineering has consistently explored metrics to measure and compare the complexity and evolution of software systems. Being able to quantify the impact of code changes in software systems is central to understanding software evolution, team productivity, and system complexity. There are many ways to quantify the impact of code changes. Among these, *Lines of code changed* (LoCC) [1] is one of the most widely used distance metrics in the software industry [2]. It is often used as a measure of software complexity, maintainability, and productivity.

While LoCC is simple to compute and interpret, it suffers from several key drawbacks that can mislead when analyzing developer effort and project activity. In this paper, we will explore the shortcomings of LoCC and propose an alternative metric: Compression Distance (CD), derived from lossless compression algorithms with large search windows. By measuring the compressibility of changes between software revisions, CD offers a novel perspective on software evolution, addressing many of the shortcomings inherent in LoCC.

Through a series of experiments, we evaluate the effectiveness of CD in quantifying code complexity, distinguishing between commit types, and mitigating biases present in LoCC.

The results suggest that CD provides a more nuanced and robust measure of software evolution, paving the way for its adoption in both academic research and industry practice. However, you need to be aware of built in biases in the metric.

## 1.1. Research Questions

This paper investigates three research questions (RQs):

- RQ1: Is the compression distance a more representative metric for quantifying the complexity of a version-controlled software repository than LoCC?
- RQ2: To what extent does compression distance discriminate between manual or semi-automatic commit types (e.g., bugfix, feature, refactoring, documentation, style)?
- RQ3: Does compression distance suffer from the same limitations as LoCC in quantifying the contributions of developers?

## 1.2. Contributions & Paper Organization

We make the following contributions:

- We define Compression Distance (CD), a distance metric based on lossless compression, and derive its per-commit delta ($\Delta$CD).
- We implement CD computation as API endpoints in the Git Truck analysis tool, leveraging ZStandard with a 2 MB search window.
- We empirically evaluate CD on two projects (Git Truck and Commitizen), showing varying correlation with LoCC, improved discrimination of commit types, and distinct author-level insights.

The remainder of the paper is organized as follows. Section 2 reviews LoCC and its limitations. Section 3 presents our proposed CD metric and its theoretical foundation. Section 4 details the methodology: data collection, metric computation, commit classification, and statistical analyses. Section 5 reports results for RQ1-RQ3. Section 6 discusses implications, practical

considerations, and limitations. Finally, Section 7 concludes and outlines directions for future work.

# 2. Background

## 2.1. Information distance and Similarity Distance Metrics in Software Engineering

In the field of information theory, the concept of *information distance* is used to quantify the similarity between two objects [3]. This is done by measuring the amount of information needed to transform one object into another using a mathematical function $F$. The most common way to measure this distance is by using a *distance metric*, which is a function that quantifies the difference between two objects.

While such a function $F$ only exists in theory, we can still approximate the information distance between two objects using various practical techniques such as diffing and compression algorithms, as we will explore in this paper.

### 2.1.1. LoCC as a measure of information distance

Since many projects utilize version control systems, such as Git [4], for keeping track of changes, we can track LoCC over time using diffing algorithms. The LoCC metric is typically defined as the number of lines added and removed in a commit. This provides a measure of the information distance between revisions of a software system. Git includes this functionality by default using the numstat argument. [5]. This is a commonly used technique used to detect activity in software systems over time [6]. It can be used to assess team velocity, developer productivity and more. These metrics can be automatically obtained via version control systems using tools like Git Truck[7]. LoCC is a useful metric for quantifying contributions or regions of interest in software systems over time and tools like Git Truck have shown the effectiveness of LoCC in the analysis of software evolution [8], [9].

### 2.1.2. Shortcomings of LoCC

There are however multiple problems when relying on LoCC as a sole metric of productivity.

#### 2.1.2.1. First problem: Ambiguous LoCC definitions

Firstly, the term itself is ambiguous and subjective to the formatting of the code. One could say that line breaks are just an arbitrary formatting character. In reality, the program could have been written in a single line of code.

See the following ambiguous examples in Listing 1. A few questions arise: should these snippets be counted as three separate lines of code or collapsed into a single line? In terms of actual work, a developer who writes either version is equally productive, yet if we simply count physical lines changed, the author of the first listing would be credited with three times the contribution of the second. This discrepancy shows how formatting alone can skew LoCC-based distance measures, as trivial style differences inflate the perceived distance between revisions and undermine the metric's reliability.

```
if (foo) {
  bar();
}
```

```
if (foo) { bar(); }
```

```
if (foo) bar();
```

Listing 1: Three semantically equivalent code snippets with different physical line counts

#### 2.1.2.2. Second problem: Rename-detection pitfalls

The LoCC metric can be distorted also by file renames, which may artificially inflate contribution counts, since renaming a file requires little effort but appears as significant line changes. Git supports rename detection using a similarity threshold[1], comparing deleted and added files to identify likely renames, utilized by tools such as Git Truck. While Git doesn't exclude renames by default, this tracking can be used to filter them out manually. However, rename detection is inherently ambiguous — at what point should we stop treating a change as a rename and instead count it as having deleted the old file and added a new one?[2]

The issue is further exacerbated when developers squash commits, potentially losing rename information. Ultimately, it's a trade-off between inflating the impact of trivial renames and missing substantial, legitimate contributions.

#### 2.1.2.3. Third problem: Automation-driven LoCC spikes

Another case where the LoCC metric falls short is when performing automated actions that affects a vast amount of files and leads to spikes in line changes. Examples of this include running formatting and linting scripts or automated lock file updates when installing packages, all of which can lead to astronomical amounts of line changes. These kinds of changes do not directly reflect genuine development effort, but they still result in high LoCC values from which one often draw this conclusion. As an illustrative example, see Figure 1 for a visualization from the tool Git Truck showing accumulated line changes per file. Notice how the file "package-lock.json" is dominating the rest of the files, due to it being an auto generated file that is modified automatically by package managers in the JavaScript ecosystem. This is an example of line changes that should to be disregarded.

[1] https://git-scm.com/docs/git-log#Documentation/git-log.txt-code-Mltngtcode°
[2] https://en.wikipedia.org/wiki/Ship_of_Theseus°

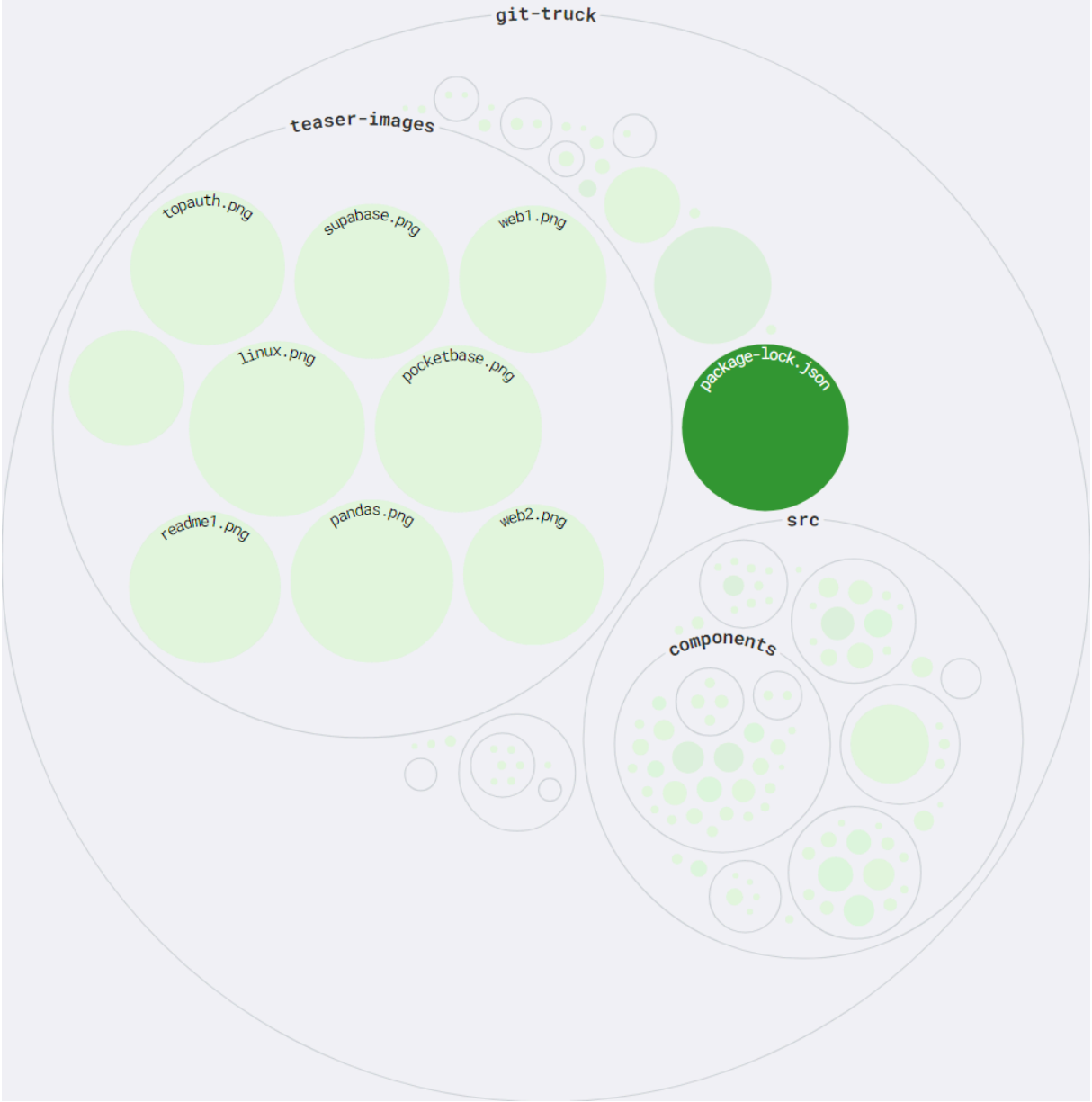


Figure 1: Screenshot of Git Truck [10] visualizing accumulated line changes per file

If the developers practice good Git hygiene (such as keeping commits small and focused, not using squash merging, and writing descriptive commit or conventional commit messages[3]), you can perform filtering to focus on for example commits fixing bugs, refactoring or implementing features, but in reality, not all teams conform to these practices.

This shows why this metric cannot stand alone as a measure of productivity. This may cause the results to be misinterpreted, overstating the activity levels of certain developers or areas of the system.

#### 2.1.2.4. Fourth problem: Survivorship bias

Another problem with the LoCC metric is that it is *not* time biased. This means that if a developer made a lot of changes to a file in the past, but then later removed all of them, they would still be credited for all of those changes, even though they are not present in the current version of the file. This is a problem when trying to analyze the history of a project and

[3]Some projects conform to conventional commit messages using tools like Commitizen [11]

understand how it has evolved over time. Tools like Git Truck track LoCC through the entire history of a project by default, weighing ancient changes as much as recent changes. In the development of Git Truck, this was attempted to be mitigated by looking at blame information[4] instead and only considering how the files that are still present in the system as of today has changed through time. However, this technique is still prone to the errors introduced by the problem stated in Section 2.1.2.2. Another attempt to solve this was the introduction of the time range slider[5], which led the user select a duration of time to analyze and ignore past changes in analysis.

[4] https://github.com/git-truck/git-truck/commit/12582272b5854d6bf23706b292f3519750023fdd°
[5] https://github.com/git-truck/git-truck/pull/731°

# 3. Approach

To mitigate the explored problems with LoCC, we can use a different approach to measure the distance between revisions of a software system. Instead of relying on the number of lines changed, we can use a metric derived from lossless compression algorithms, to measure the distance between revisions.

## 3.1. Mitigating the problem of ambiguous definitions of LoCC and rename-detection pitfalls

To mitigate the ambiguity described in Section 2.1.2.1, we can instead consider the change in the number of bytes instead of LoCC as a quantifier of developer activity.

However, this means we can no longer rely on line-based diffing algorithms, akin to those used in Git [12] and must use an alternative method to measure the distance between revisions.

We can mitigate the problem with ambiguity and renames by concatenating all the files existing in each commit into a single file buffer. We will refer to this as the concatenated commit buffer, CCB. We can measure its size before and after the commit to calculate a **byte distance metric**, see Equation 1.

$$\Delta \mathrm{D}(x) = |x| - |x - 1| \tag{1}$$

where $\Delta \mathrm{D}(x)$ is the distance in bytes, $|x|$ is the size of the given commit buffer and $x - 1$ is the size of the previous commit buffer.

This method does not address the problem with automatic actions overstating developer impact.

This leads us to the Compression Distance.

## 3.2. Mitigating the problem of automation-driven LoCC spikes

Instead of measuring the static distance in bytes before and after the commit, we can instead try to measure the compression of the changes we added. Using a lossless compression algorithm, we compress the concatenation of the given commit buffer $x$ with the newest revision of the project $y$ and compare this value with the one for the previous commit $x - 1$. The hypothesis being that this would make it such that large repetitive actions would have a lower impact, since they would compress better than smaller, but more complex changes. This assumes that the changes added in a commit will compress better in the presence of similar code. This requires the compression algorithm to have search window larger than double the size of the project files we intend to analyze, as explored in [13].

## 3.3. Definition of Compression Distance

We define the Compression Distance CD metric as a measure of how much a given concatenated commit buffer compresses with the baseline commit buffer:

$$\mathrm{CD}(x, y) = |C(x)| - |C(x \cup y)| \tag{2}$$

where CD is the compression distance, $x$ is the given concatenated commit buffer and Z is a lossless compression algorithm.

Now we can define the impact of the commit $\Delta\mathrm{CD}$, compared to the previous commit buffer, giving us

$$\Delta\mathrm{CD} = \mathrm{CD}(x) - \mathrm{CD}(x-1) \tag{3}$$

Unlike the normalized compression distance, this metric is not bounded and is dependent on the absolute sizes of the compressed files. This is fine in this scenario, as we want to compare commits in the same project and the magnitude of the compression distance tells us an interesting story about the impact of the commit.

Then, we can compute the change in the size in bytes before and after the commit.

If we attempt to compress the commit buffers in presence of the newest commit buffer of the repository, we get the side effect of introducing survivorship bias into the system. By measuring the compressed distance to the newest commit buffer of the project, we value changes that are more akin to the newest version higher, in order to tell a story about how we got to the newest version and which commits were the most influential in getting there. For some purposes this makes sense, as long as you are aware of survivorship bias. For example, a detour in the project that didn't make it in the newest version is weighted by a negative distance, telling us this brought the project further away from the newest version, but it might still have been a valuable journey to take with the project. See Figure 2 for a diagram showing how some commits increase the distance from the final version.

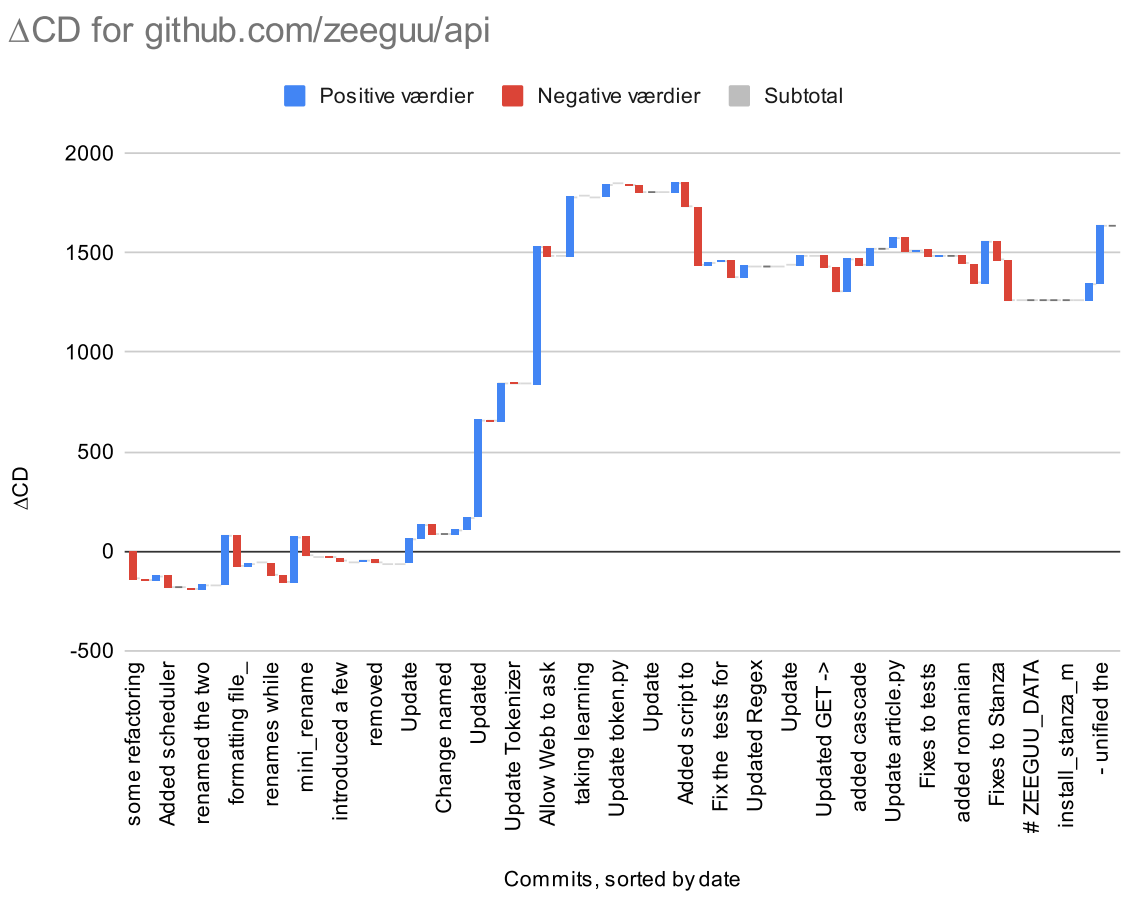


Figure 2: Waterfall diagram showing ΔCD for the latest 100 commits in the repository github.com/zeeguu/api°

## 3.4. Mitigating survivorship bias in compression distance

If we intend to value these types of detours better, we can use the magnitude of the distance instead, which then gives us the impact of the commit, no matter if it moved the project closer or further from the project.

To mitigate these problems, we can use an alternative approach where we concatenate the entire state of the repository, makes the analysis resistant to renames, as we don't care about file names, we only care about the bytes contained in the commit. However, in this case, we can no longer rely on diffing the concatenated string. We need to use another approach to derive the information distance from before and after a commit.

## 3.5. Normalized Compression Distance

Normalized Compression Distance (NCD) [14] is a effective way of measuring the similarity between two compressible objects, using lossless compression algorithms such as ZStandard. It is a way of approximating the Normalized Information Distance and has widespread uses, such as clustering analysis [15].

According to [13], NCD is an effective distance measurement, as long as it is used properly, which can be achieved if you understand how compression algorithms work. Most compression algorithms use a sliding window of context from which to compare new data to old data,

in order to limit memory consumption. If the size of this window is too small, matches might not be found and the data might not compress as well. Since we are using the compression to quantify the similarity, it is crucial that we use a compression algorithm with support for a window size that exceeds the size of the data that we want to compare. In fact, if we compare two version of an object to determine how it has changed over time, we need a window size twice the size of the largest version of the object, in order to not overlook shared patterns.

However, nobody has used NCD based distance metrics in version control system analysis yet. The hypothesis of this project is that NCD-derived metrics could function as a complement to the more well-known LoCC metric, and be resilient to the problems that LoCC suffers from.

# 4. Methodology

## 4.1. Data collection

During this project, the analysis tools were implemented as API endpoints exposed in the Git Truck project [16]. This was done due to the foundation for performing git analysis were already readily available in Git Truck project, which sped up the development of the tools.

We used these endpoints to collect and visualize data about different repositories to draw conclusions about the Compression Distance metric.

### 4.1.1. Repository Selection

For the analysis of this project, two differently sized projects were chosen. Git Truck was chosen, as we were familiar with its history and knew that an attempt was made to conform to good practices of making git commits, such as using the imperative mood etc., which made categorizing the commits simpler. Commitizen was chosen as well, due to its nature of conforming to the use of Conventional Commits [17], which automatically could be categorized.

The different versions of Git Truck where chosen due to the known time periods on which the project was worked on by a previously known set of contributors.

From Table 1, we see that all the chosen repositories lie below the maximum of 1MB, which ensures that sliding search window of the compression algorithm is able to take the repositories in their entireties into account, as mentioned in Section 3.5.

| | Newest commit | | |
|---|---|---|---|
| **Repository@revision** | **Hash** | **Buffer size** (MB) | **Commits** |
| Git Truck@ncd [10] | bf46e09 | 0.365 | 1356 |
| Git Truck@v2.0.4 [10] | d385ace | 0.318 | 1260 |
| Git Truck@v1.13.0 [10] | 71ae30d | 0.259 | 1242 |
| Commitizen@master [11] | e177141 | 0.771 | 1977 |

Table 1: Git repositories and revisions that were chosen for this project and some of their properties

### 4.1.2. File Inclusion/Exclusion

We chose to include/exclude certain file extensions, to focus the results on code files.

In general, file extensions commonly associated with code were included, while binary files like images, videos, and audio files as well as miscellaneous files were excluded [18].

If any extensions were found that were neither included nor excluded, an automatic warning was reported in the console, in order to consider whether it should be included or excluded.

### 4.1.3. API Data retrieval

The tool is able to go through a specified range of the history of a git repository and compute metrics for each commit.

The current version of the tool is as of writing not published yet and has to be run manually by cloning the source code.

Queries can be performed using query parameters like so: Among other metrics, the endpoint is able to compute the Compression Distance in relation to the baseline commit, the newest commit at the given revision in the repository.

- Repository can be specified using the repo=<folder> parameter[6]
- Baseline branch or revision is specified with the branch=<revision> parameter.
- Amount of commits to analyze is specified using the count = $N$|Infinity parameter, going backwards from branch[7]

To generate the data for this project, the following queries were used:

| Repository | Branch | Count |
|---|---|---|
| git-truck | ncd | Infinity |
| git-truck | v2.0.4 | Infinity |
| git-truck | v1.13.0 | Infinity |
| commitizen | master | Infinity |

Table 2: Parameters used to gather the analysis data

This produces a CSV output that can be processed in a data processing tool. During this project, Google Sheets was used for this purpose. Useful information and progress updates is reported in the console.

## 4.2. Metric Computation

In this section, we describe the detailed steps to compute Compression Distance (CD) for each commit. The process is divided into four sub-sections:

### 4.2.1. Concatenated Commit Buffer (CCB) Construction

We begin by constructing a concatenated commit buffer (CCB) for every commit, reading all the file contents into a single buffer. The contents of the files are obtained via the `git cat-file <blob hash>` command and reading the standard output of the command.

### 4.2.2. Compression Setup

Next, we compress each CCB and its paired baseline buffer using ZStandard (zstd) [19].

[6]The repo parameter refers to a specific git repository folder located relative to where the tools was downloaded.

[7]Passing Infinity as count makes the tool go through all the commits in the given repository.

The ZStandard compression algorithm (zstd) [20], has a window log of 21 [19] for its default 3 level compression, making the window size $2^{21} = 2\text{MB}$. This makes zstd suitable for this task, as long as you are aware of the limit and remember to adjust it as needed.

We used the default compression level of 3, which has a window size of $2^{21}B = 2$ MB.

We checked that the repositories we used were smaller than half of this size[8], in order for the compression algorithm to consider the entire commit buffer when attempting to compress it.

#### 4.2.3. Calculating CD & $\Delta$CD

We then calculate the Compression Distance using Equation 2. Then, to get the compressed distance contribution for each commit, we simply subtract it with the compression distance from the previous commit as seen in Equation 3.

This method adds intentional survivorship bias into the metric, favoring code that is more easily compressible with the final version.

## 4.3. Commit Classification

Using keyword searching, we were able to categorize the many of the commits automatically. For the Commitizen repository, all the commits were automatically categorized due to the nature of conforming to Conventional Commits. We used the following keywords for categorizing the commits:

For Commitizen, we used: Test, Fix, Feat, Refactor, Style, Docs, as

For Git Truck, we used: Bump, Refactor, Fix, Feature.

## 4.4. Statistical Analysis

We evaluated whether per-commit $\Delta$CD aligns with traditional complexity measured by LoCC. For each repository, we created log-log scatter plots with linear trend-lines and computed the coefficient of determination, $R^2$, between $\Delta$CD and LoCC across all commits.

[8]Due to compressing the newest commit buffer with each commit, meaning we need at least 2x the baseline buffer, assuming that no commit buffer is larger than the final buffer, but there is still some wiggle room for all of the projects, as none of them surpass 1 MB

# 5. Results

In this section, we will explore whether we have answered the research questions presented in Section 1.1, provided here for convenience:

- RQ1: Is the compression distance a more representative metric for quantifying the complexity of a version-controlled software repository than LoCC?
- RQ2: To what extent does compression distance discriminate between manual or semi-automatic commit types (e.g., bugfix, feature, refactoring, documentation, style)?
- RQ3: Does compression distance suffer from the same limitations as LoCC in quantifying the contributions of developers?

The data used to answer these questions is published in Google Sheets [21], [22].

## 5.1. RQ1: LoCC vs. $\Delta$CD Correlation

To answer RQ1, we will explore newest version of each repository and see how the CD metric correlates with LoCC.

### 5.1.1. Correlation Results

| Project | Revision | $R^2$, linear | $R^2$, power |
|---|---|---|---|
| Git Truck | ncd | 0.825 | 0.494 |
| Commitizen | master | 0.297 | 0.506 |

Table 3: Correlation between LoCC and $\Delta$CD

From Table 3, it seems that for Git Truck, there exists a linear correlation between LoCC and $\Delta$CD, however for Commitizen, a power regression produces a better correlation.

## 5.2. RQ2: Discrimination Across Commit Types

### 5.2.1. $\Delta$CD Distributions by Category

For this experiment, we will look at the Commitizen repository. If we plot each category of commit as a series on a log-log scatter-plot, we can see some interesting patterns. See Figure 3.

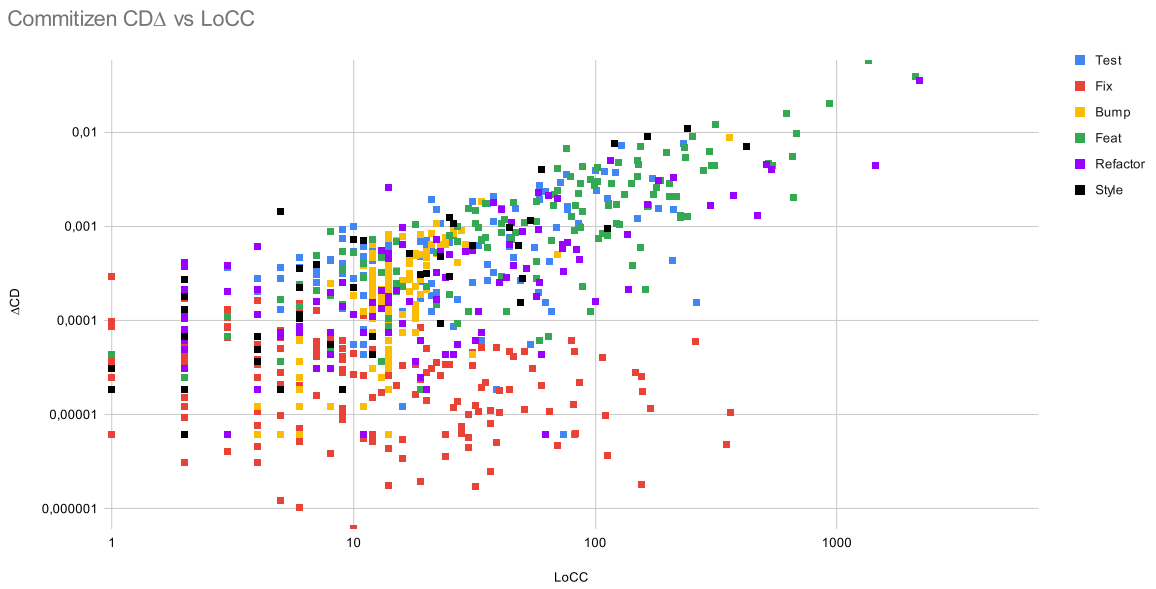


Figure 3: Log-log scatter-plot of commits in the Commitizen repository, with automatically categorized commits

From the plot, we can see that bug fix commits specifically has a tendency to have a lower $\Delta$CD metric.

We can also see that version bumps vary much less in impact compared to the other categories. Feature commits generally have a larger LoCC than other commits, and might contain more novel code that compress less, compared to bug fixes.

## 5.3. RQ3: Developer Contribution Analysis

For this experiment, we will look at the Git Truck repository across two time periods and see whether survivorship bias plays a role in $\Delta$CD.

We accumulate the byte distance, LoCC and $\Delta$CD for each developer throughout the two time periods.

We then aggregate each metric for each contributor throughout the history of the repository.

For context, Git Truck was initially developed by a group of four developers. Later, the project was continuously contributed to several developers, and even later Thomas contributed to the project during his their master thesis.

We will compare the project before and after the master thesis by Thomas.

### 5.3.1. Author-Level Aggregates

| AUTHOR | $\Delta$CD | LOCC |
|---|---|---|
| Jonas | 54.987% | 62.370% |
| Thomas | 21.425% | 15.719% |
| Emil | 15.793% | 13.482% |

| AUTHOR | $\Delta$CD | LOCC |
|---|---|---|
| Jonas | 47.069% | 58.908% |
| Thomas | 46.636% | 24.148% |
| Emil | 3.260% | 10.021% |

| AUTHOR | ΔCD | LOCC |
|---|---|---|
| Kristoffer | 7.124% | 7.625% |
| Mircea | 0.670% | 0.155% |

| AUTHOR | ΔCD | LOCC |
|---|---|---|
| Dawid | 3.241% | 1.856% |
| Kristoffer | 0.341% | 4.134% |

Table 4: Cumulative LoCC and ΔCD
Left: Before master thesis (v1.13.0), Right: after (v2.0.4)

See Figure 4 for the distribution over cumulative byte distance, LoCC and ΔCD over the two time periods. See Figure 5 for how the cumulative distribution has changes over time.

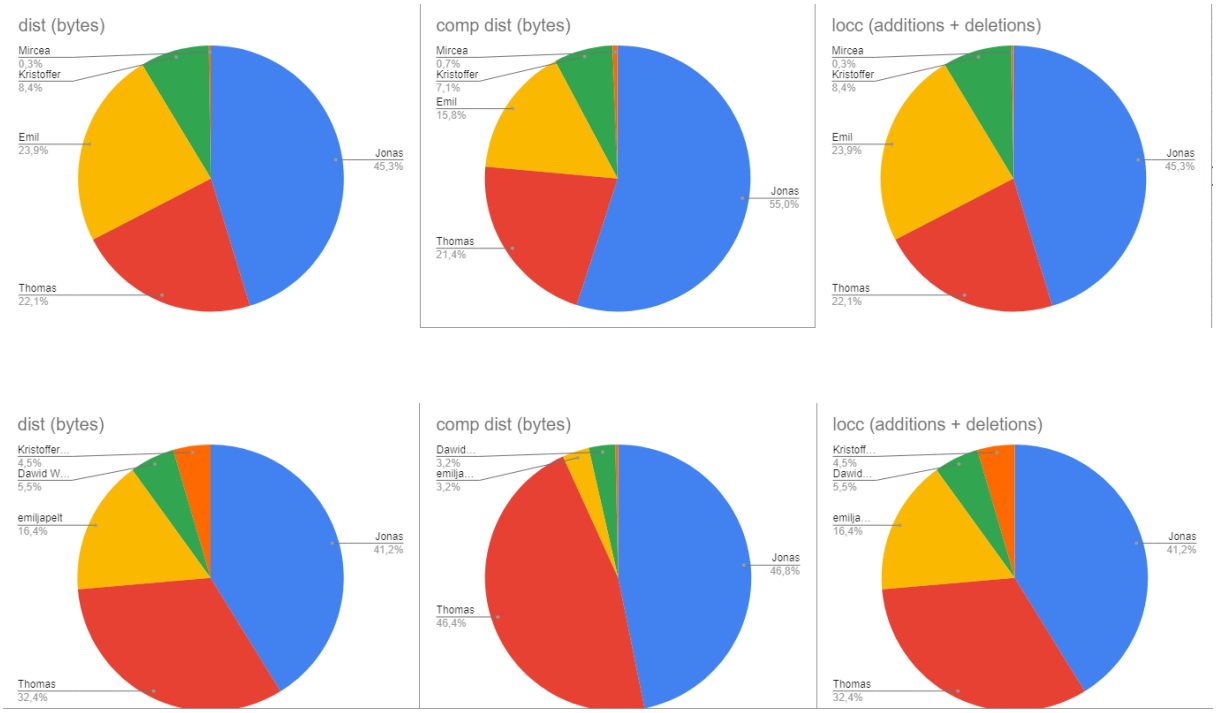


Figure 4: Pie charts of the cumulative author distribution before (top) and after (bottom) the thesis project by Thomas.
Left: Cumulative Byte Distance, Center: Cumulative ΔCD, Right: Cumulative LoCC

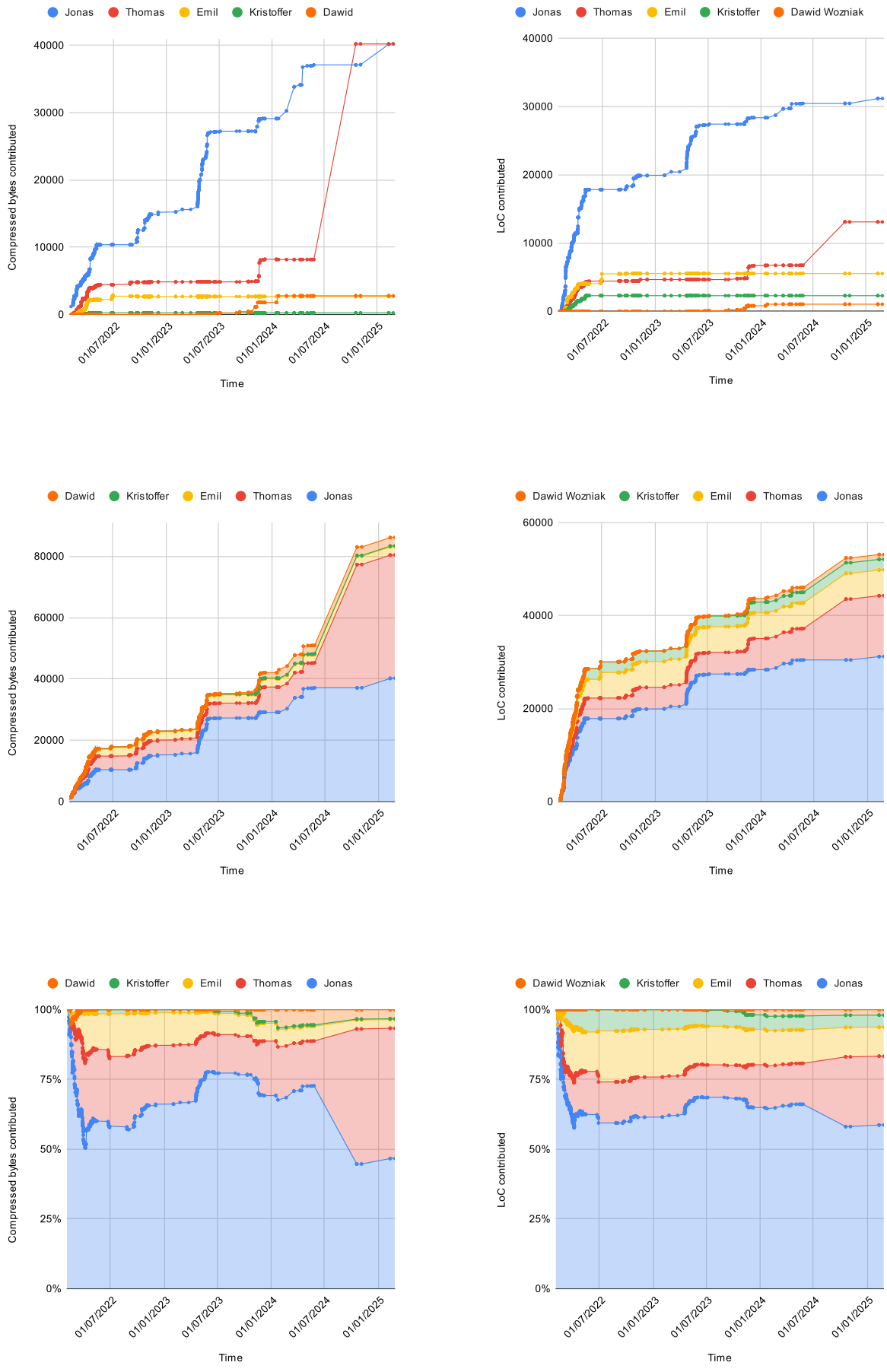


Figure 5: Cumulative area charts of the author distribution over time.
Overlap (top), stacked (middle) stacked 100% (bottom)
ΔCD based (left) and LoCC based (right)

### 5.3.2. 5.3.3 Observations: Survivorship Bias Skew

We observe that before Thomas worked intensively on the project, his contribution distributions measured in LoCC and $\Delta$CD were fairly close, however, after his thesis, his share of accumulated $\Delta$CD equalized with that of Jonas.

This is also clear to see on the area time series charts in Figure 5. We can see that using the $\Delta$CD metric, Thomas is able to surpass Jonas in the author distribution. We can also clearly see when Thomas merged his master thesis into the repository and that the merge technique was squashing all of the commits into one, hence the big spike upwards and long period of few commits. This clearly illustrates the information that is lost when developers choose squash merging over regular merges or rebase merging strategies. This is a limitation that neither LoCC, $\Delta$CD or any other metric can preserve, unless the commits are recovered by rebasing them back into the branch.

# 6. Discussion

## 6.1. Interpretation of RQ1 Findings

In Section 5.1, we observed that there was not necessarily a clear correlation between LoCC and $\Delta$CD, which tells us that the two metrics measure different things and have their own purpose.

## 6.2. Interpretation of RQ2 Findings

In Section 5.2, we observed that certain types of commits contribute more information to the codebase than others. This is intuitive, as introducing a novel feature typically adds more unique content than modifying existing code for bug fixes or version bumps.

## 6.3. Interpretation of RQ3 Findings

There are several hypothesis for the differences observed in Section 5.3. Since Thomas changed a lot of the codebase during his thesis work, the survivorship bias of the metric favor his recent changes, when judging the history of the project. Another part of the explanation might be that his thesis entailed adding a database to Git Truck, contributing a lot of SQL code to an already TypeScript dominated project, which might also explain the large spike in the $\Delta$CD attribution to Thomas. The hypothesis being that contributing SQL code to the codebase will compress worse than contributing TypeScript code.

This demonstrates the built in survivorship bias that the metric includes and illustrate how you need to be aware of this when using the metric for judging work. For judging work done in group projects, one should select time ranges that correspond to the period of work during the project.

We see that if we measure the author distribution based on cumulative $\Delta$CD contributions, we get survivorship bias built into the metric, which tells us more about who contributed the most to get the system in its current state.

## 6.4. Practical Implications

This study has shown that if implemented with caution for compression window sizes and built in bias, a metric like the compression distance has its place in the arsenal of metrics used in analyzing software evolution. It's a viable supplement to existing metrics and might especially be useful for assessing student projects, as long as detours in the project are also noticed.

## 6.5. Considerations for End users

While measuring automatic velocity is a good idea, it is not the only thing to consider. For a user facing project, it is important to also consider the user experience and the impact of changes on the end users. Some small bug fix might have a small impact on the code, but a large impact on the end user.

## 6.6. Limitations

There are however some limitations to this approach, that makes it less desirable compared to alternatives, depending on the needs.

### 6.6.1. Performance:

We've found that, at least with the current implementation, compressed distance is much slower to compute than traditional metrics like LoCC built in to Git Truck. For example, analyzing the entire history of Commitizen in Git Truck was measured to take $1.55 \pm 0.07$ seconds, while it takes $374.28 \pm 5.41$ seconds for the proposed tool in its current state, making it a $\approx 250$ times slower metric in this scenario.

### 6.6.2. Scalability

For very large repositories, it becomes unfeasible to calculate the compression distance for the entire project at once, as it requires a lot of memory and processing power. This is due to the fact that the compression algorithm needs to keep track of the entire state of the repository in memory. The limited window size [13] leading to distorted distance measurements, especially in worst case when comparing identical objects exceeding these size constraints. This means that we are back to just measuring something similar to the byte distance.

### 6.6.3. Biases

The built in survivorship bias is included with this metric for better or for worse. Alternative algorithms were tested during the project, but did not show promise and were abandoned early on.

# 7. Conclusion & Future Work

## 7.1. Summary of Key Contributions

This paper has presented a method for using compression as a metric for information distance, defined as

$$\mathrm{CD}(x, y) = |C(x)| - |C(x \cup y)| \tag{4}$$

where $C$ is a compression function, $x$ is the concatenated commit buffer and $y$ is the baseline concatenated commit buffer.

The results suggest that CD provides a more nuanced and robust measure of software evolution, paving the way for its adoption in both academic research and industry practice. The results also show that you need to be aware of built in biases in the metric.

## 7.2. Future Work

There are many directions to take this project further, including exploring usability, scalability, performance and bias elimination.

### 7.2.1. Usability

Incorporating the tool into Git Truck would make it so others could experiment with the data as well. It would also be interesting to see how this data could be incorporated and presented into the visualizations of Git Truck. It could work as an extension of the author distribution already found in Git Truck, that is currently based on LoCC.

### 7.2.2. Scalability

Attempting to run the tool on a large repository, like the source code for Linux [23] yields a maxBuffer exceeded error:

```
RangeError: stdout maxBuffer length exceeded
    at Socket.onChildStdout (node:child_process:481:14)
    at Socket.emit (node:events:507:28)
    at addChunk (node:internal/streams/readable:559:12)
    at readableAddChunkPushByteMode (node:internal/streams/readable:510:3)
    at Socket.Readable.push (node:internal/streams/readable:390:5)
    at Pipe.onStreamRead (node:internal/stream_base_commons:189:23) {
  code: 'ERR_CHILD_PROCESS_STDIO_MAXBUFFER',
  cmd: 'git ls-tree -r HEAD'
```

This shows that future work could be done to investigate whether this metric could scale to very large repositories.

### 7.2.3. Performance

Future work could be done to investigate methods of speeding up the computational process. The process might be parallelizable and could utilize using shared memory with dynamic garbage collection, to reduce the memory overhead of the analysis. The current approach caches file buffers, but leaves garbage collection to the runtime.

### 7.2.4. Bias elimination

Attempting to design an algorithmic formula for avoiding the survivorship bias included in the metric might be an interesting project to undertake, if the bias is deemed undesirable to the use case.

# 8. Acknowledgments

The git analysis pipeline powering Git Truck was used as a foundation for developing the analysis tool.

We would like to thank Christian Gram Kalhauge <chrg@dtu.dk> for proposing the idea for this project and for his valuable external supervision and guidance.

```
while (workingOnThisProject) {
  Jonas.becameAFather();
}
```

# Bibliography


[1] "Source lines of code - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Source_lines_of_code

[2] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm, "A SLOC counting standard," in *Cocomo ii forum*, 2007.

[3] "Information distance - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Information_distance

[4] "Git." [Online]. Available: https://git-scm.com/

[5] "Git - git-log Documentation." [Online]. Available: https://git-scm.com/docs/git-log#Documentation/git-log.txt-code%E2%80%93numstatcode

[6] M. Goeminne and T. Mens, "Analyzing ecosystems for open source software developer communities," *Software Ecosystems*. Edward Elgar Publishing, pp. 247–275, 2013.

[7] K. Højelse, T. Kilbak, J. Røssum, E. Jäpelt, L. Merino, and M. Lungu, "Git-truck: Hierarchy-oriented visualization of git repository evolution," in *2022 Working Conference on Software Visualization (VISSOFT)*, 2022, pp. 131–140.

[8] M. Lungu, R.-H. Pfeiffer, M. D'Ambros, M. Lanza, and J. Findahl, "Can git repository visualization support educators in assessing group projects?," in *2022 Working Conference on Software Visualization (VISSOFT)*, 2022, pp. 187–191.

[9] A. Neyem, J. Carrasco-Aravena, A. Fernandez-Blanco, and J. P. Sandoval Alcocer, "Exploring the Adaptability and Usefulness of Git-Truck for Assessing Software Capstone Project Development," in *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1*, 2025, pp. 847–853.

[10] "git-truck/git-truck: Git repository visualizations, cumulative contribution statistics and more. Run npx -y git-truck to use Git Truck today!." [Online]. Available: https://github.com/git-truck/git-truck

[11] "commitizen-tools/commitizen: Create committing rules for projects auto bump versions and auto changelog generation." [Online]. Available: https://github.com/commitizen-tools/commitizen/

[12] "Git - diff-options Documentation." [Online]. Available: https://git-scm.com/docs/diff-options/2.6.7#Documentation/diff-options.txt%E2%80%94diff-algorithmpatienceminimalhistogrammyers

[13] M. Cebrián, M. Alfonseca, and A. Ortega, "Common pitfalls using the normalized compression distance: What to watch out for in a compressor," 2005.

[14] "Normalized compression distance - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Normalized_compression_distance

[15] R. Cilibrasi and P. Vitanyi, "Clustering by compression." [Online]. Available: https://arxiv.org/abs/cs/0312044

[16] J. N. Røssum, "git-truck/src/routes at ncd · git-truck/git-truck." [Online]. Available: https://github.com/git-truck/git-truck/tree/ncd/src/routes

[17] "Conventional Commits." [Online]. Available: https://www.conventionalcommits.org/en/v1.0.0/

[18] "git-truck/src/routes/get-commits.tsx at ncd · git-truck/git-truck." [Online]. Available: https://github.com/git-truck/git-truck/blob/ncd/src/routes/get-commits.tsx#L11-L89

[19] "zstd/lib/compress/clevels.h at f9938c217da17ec3e9dcd2a2d99c5cf39536aeb9 · facebook/zstd." [Online]. Available: https://github.com/facebook/zstd/blob/f9938c217da17ec3e9dcd2a2d99c5cf39536aeb9/lib/compress/clevels.h#L31C7-L31C9

[20] "facebook/zstd: Zstandard - Fast real-time compression algorithm." [Online]. Available: https://github.com/facebook/zstd

[21] J. N. Røssum, “Research Project Data - Google Sheets.” [Online]. Available: https://docs.google.com/spreadsheets/d/1Hiz25Vg-z2_KGzX_R-oCP9w37zgY1qNvn4A_f0CVbFE/edit?gid=2027480796#gid=2027480796°

[22] J. N. Røssum, “Subjective commits - Google Sheets.” [Online]. Available: https://docs.google.com/spreadsheets/d/1Yp9H2HCJs9Kl_fB4I6X4odfTXb3l4l04NdDGXNdWXrk/edit?pli=1&gid=1689528729#gid=1689528729°

[23] “torvalds/linux: Linux kernel source tree.” [Online]. Available: https://github.com/torvalds/linux°

# Index of Figures



# Index of Tables



# Index of Listings