A Framework for Automating the Measurement of DevOps Research and Assessment (DORA) Metrics

Brennan Wilkes
University of Victoria
Victoria, Canada
brennan@codexwilkes.com

Alessandra Maciel Paz Milani
University of Victoria
Victoria, Canada
amilani@uvic.ca

Margaret-Anne Storey
University of Victoria
Victoria, Canada
mstorey@uvic.ca

***Abstract*—The DevOps Research and Assessment (DORA) metrics have been widely accepted by the software industry as a powerful method to quantify DevOps performance, leading to significant interest in their measurement. Existing proprietary solutions are highly customised, and require specific types of cloud infrastructure, limiting their suitability for projects such as libraries, frameworks, and open source projects. To address this gap, we present a framework which operationalizes the DORA metrics independently of a project's software development life-cycle or type of deployment. We demonstrate the general applicability of this framework by using it to calculate the throughput and stability of 304 popular open source repositories. We find that the time-series data it produces provides meaningful insights into the trending direction of a project's recent and retrospective throughput and stability performance, especially when significant changes in metrics are correlated with major events in the project's history. We conclude with recommendations for augmenting our approach with additional information such as bug criticality when such information is available.**

***Index Terms*—Technological, DevOps, DORA, Metrics, OSS**

I. Introduction

Throughout the previous two decades, software teams have increasingly prioritised software stability and developer productivity [1]. This prioritisation has led to DevOps, a cultural and professional movement that aims to bridge the gap between software development and IT operations [2]. The DevOps methodology aims to reduce the time and effort required to bring new software and services to market through the use of tools and techniques such as CI/CD (continuous integration and continuous delivery), infrastructure as code (IaC), automated testing, and agile development life-cycles. Using a modern version control system such as git is a common DevOps practice [3], and as such, most successful projects use a git repository (or equivalent internal version control tool) to store source code versions [4]. We leverage this common dependency when designing our framework. Today, many companies and projects assess their DevOps practices and drive change based on metrics such as the DORA metrics [5].

The DORA metrics were created by Nicole Forsgren, Gene Kim, and Jez Humble to measure and compare DevOps performance across organizations and are described in detail in their book, Accelerate [6]. They were further popularized by the annual State of DevOps Report [7], more recently written by staff at Google. Accelerate [6] and the State of DevOps Reports [1] have shown four metrics correlate with IT performance: *deployment frequency*, *lead time for changes* (these first two measure throughput), *mean time to recover*, and *change failure rate* (these last two measure stability). The DORA group has used data from 33,000 survey respondents [7] to publish reported industry averages for each of these metrics in the annual State of DevOps report. In recent years, the software industry has widely adopted the DORA metrics to measure software delivery and IT performance [4]. In a 2020 survey of 500 software professionals, Atlassian [5] found that 93% of survey respondents believed DevOps performance had a clear impact on business performance.

Despite popularizing the DORA metrics, neither Accelerate nor the State of DevOps Report describe reproducible methods for measuring a given project's metrics. Existing approaches using telemetry data are varied [8], but often make assumptions that exclude many types of software projects (such as open source ones), while proprietary solutions hide implementation details. Indeed, the 2020 Atlassian DevOps Trends Survey [5] found that while 95% of survey respondents believed measuring DevOps performance with the DORA metrics was important, 51% did not have an effective way to do so, and 54% found it difficult to measure the effect of DevOps progress. Existing solutions such as the Four Keys Project [1] or GitLab DORA Metrics [2] impose restrictive requirements on projects such as the usage of Azure DevOps or GitLab Incidents, existence of cloud infrastructure and corresponding incident monitoring, or monitored build pipelines. These restrictions prevent significant sub-communities of the greater software industry from being served (such as open source projects like libraries, frameworks, and programming languages), and prevent fair, reproducible comparisons between the metrics of different types of projects.

The State of DevOps Reports [1] show the universal applicability of the DORA metrics to projects of all types across the entire software industry. They further provide performance *clusters* for organizations to compare their internal measurements against DORA's survey results, but a lack of standardization in the calculation of said metrics makes accurate benchmarks difficult. The inability to accurately compare the

[1]https://github.com/dora-team/fourkeys
[2]https://docs.gitlab.com/ee/user/analytics/dora_metrics.html

metrics of different projects excludes these solutions for use in research, and makes it difficult for organizations to judge their own performance. Additionally, current solutions cannot accurately compare current performance to past performance, making retroactive analysis or analysis of trend direction difficult.

In our paper, we present a framework which operationalizes the DORA metrics with significantly less restrictive requirements about project structure and software development processes. We propose that this framework can be used to automatically measure the DORA metrics of a broad range of projects across the software industry, both in real time and retroactively. We show how these measured metrics can be used to compare the DevOps performance of multiple projects, and discuss alternative designs which can be used when additional information is available.

The contributions from our research are as follows:

- A project-agnostic **framework** and implementation for automatically measuring the DORA metrics.
- A **dataset** that includes the measured DORA metrics for 304 open source projects hosted on GitHub with comparisons to the State of DevOps Report benchmarks.
- **Future research direction** that builds on the insights gained in this study about the applicability and limitations of automatic DORA metric measurement.

The rest of our paper is organized as follows. Section II presents the background behind the DORA metrics and how we can mine GitHub to operationalize the DORA metrics. Section III describes our framework, and defines the key concepts we use and how we calculate the DORA metrics. Section IV shows how the framework can be applied to a sample of open source projects, and describes the insights from applying the framework to this sample, including the illustrative stories of three sample repositories. Section V covers a discussion of the main insights we achieved, along with future research directions. Section VI discusses threats to validity. Finally, Section VII summarizes the implications of this work.

II. Background

In the following section we describe the four key DORA metrics, and allude to how we will operationalize them so they can be more broadly applied. We also provide background information on how we will use git and GitHub.

A. The DORA Metrics

The DORA metrics emerged from years of research conducted by the DORA group as the most reliable indicators of DevOps performance. They consist of four metrics to measure an organization or projects software delivery performance, divided into two metrics for throughput, and two metrics for stability.

1) Throughput Metrics: Deployment frequency is a key metric that measures how often an organisation deploys new software or updates existing software, and is often measured as the number of deployments per year. A high *deployment frequency* indicates that an organisation is able to quickly and reliably deliver new software and updates, while a low *deployment frequency* may signal that the organisation is struggling to keep up with the demand for new features and changes. When analysed in conjunction with *lead time for changes*, *deployment frequency* can represent a team's throughput and the rate at which they can deliver value to users. Strong team throughput can improve customer satisfaction and drive business growth [6]. *Lead time for changes* measures the time difference between software being written and that same software being deployed to users, and is often measured in days. In practice, this means the difference between a git commit being committed, and that commit being deployed to production. A high *lead time for changes* often represents bottle necks in the software delivery pipeline.

2) Stability Metrics: Mean time to recover measures the amount of time it takes for an organisation to restore normal service after an incident or disruption, and is often measured in days. Along with *change failure rate*, *mean time to recover* is central to software stability, which refers to the reliability and availability of an organisation's software and services. *Change failure rate* measures the percentage of deployments to an organisation's software or IT systems that fail or result in unintended consequences. This can also be considered as the percentage of deployments which contain a bug. A low *change failure rate* indicates that an organisation is able to successfully implement changes to its software and systems without causing disruptions or issues, while a high *change failure rate* may indicate that the organization is struggling to effectively manage and deploy changes.

3) DevOps Performance Clusters: The annual State of DevOps Report [1] surveys software practitioners on their organizations software delivery and IT operations practices. This data is then analyzed using a latent class analysis [9] to produce an optimal number of clusters [10] which best classify the dataset. These clusters are then labelled *low*, *medium*, *high*, and *elite*. (Depending on the specific report, the *elite* cluster is sometimes a subset of the *high* cluster, and is sometimes omitted entirely.) DORA describes organizations classified as part of the *high* and *elite* clusters as those organizations that effectively use DevOps practices such as continuous delivery, cloud technologies, and automation, while organizations that are classified as members of the *low* performance cluster, tend to work in silos, use on-premise infrastructure, and are resistant to operational change. The report also publishes a benchmark chart for readers to compare third party organizations to each of the performance clusters. This chart describes the range of each of the DORA metrics for each cluster, and is shown in Table I. We will use this set of benchmarks to compare our data to the clusters defined in the State of DevOps Report.

B. Git and GitHub

In our study, we consider open source projects that use git and are hosted on GitHub.

TABLE I
2021 STATE OF DEVOPS REPORT CLUSTER BENCHMARK [1]

DORA Metric	**Performance Cluster**			
	Elite	**High**	**Medium**	**Low**
Deployment Frequency	Multiple per day	Once per week to once per month	Once per month to twice per year	less than twice per year
Lead Time for Changes	Less than one hour	One day to one week	One to six months	More than six months
Mean Time to Recover	Less than one hour	Less than one day	One day to one week	More than six months
Change Failure Rate	0%-15%	15%-30%	15%-30%	15%-30%

Internally, git organizes commits into a directed acyclic graph [11], referred to as the commit graph or commit tree, where commits are represented as nodes. Each commit (besides the root) has exactly one parent, and git stores a directed edge to that parent. This structure allows project contributors to track the evolution of the project codebase over time, and enables powerful features such as branching and merging. One algorithm which takes advantage of this structure is the SZZ algorithm.

Introduced by Śliwerski et al. [12], SZZ aims to link a bug-fixing commit to a bug-inducing commit to identify the source of a bug. To do this, it traverses the git commit graph of a project backwards from the bug-fixing commit, examining each commit with a diff tool. It compares each of these commits to the bug-fixing commit until it finds one or more commits which match the lines changed in the bug-fix. Despite being the standard for bug identification, SZZ has been shown to miss bug-inducing commits, as well as mislabel regular commits as bug-inducing [13]. Numerous improvements have been suggested, including ignoring whitespace and comments [14]. We will use SZZ, along with the recommended improvements, in our calculation of *change failure rate*.

In the open source community, an increase in the use of DevOps practices has led to more reliance on GitHub [15]. GitHub is a web-based platform for hosting and collaborating on software projects. It hosts a project's git repository and provides tools and features that support the DevOps approach, such as CI/CD, issue tracking, and a limited project management platform. Other platforms, such as GitLab and Bitbucket, offer similar features.

On GitHub, "issues" are created by project contributors as a way to represent and track bugs, feature requests, and other project tasks. GitHub issues have labels, which are tags used to categorize issues of similar types. For example, an issue might be labelled "bug" if it is a bug report or "enhancement" if it is a feature request. Issues can also be assigned different states, such as "closed as completed", "closed", or "open", to help developers keep track of the status of each task. Issues are closed when the work associated with them no longer needs to be done. There are multiple reasons an issue can be closed, such as when the underlying bug has been fixed, the requested feature has been completed, or the project team has decided that the issue is not important or is invalid. The "close" event can sometimes be triggered directly by a commit which uses a descriptive commit message, such as "Closes #138", or by the "merged" event of a linked pull request.

Whenever an event related to an issue occurs, this event is linked to the issue by its timeline. These events include comments, labels being added or removed, state changes, or mentions from pull requests. The issue timeline is a useful tool for developers as it provides a comprehensive overview of the issue's entire lifecycle, making it easier to understand the context of the issue and how it was resolved.

Next we describe the framework design and how we leverage the above features of git and GitHub for calculating the DORA metrics of projects hosted on GitHub.

III. A FRAMEWORK FOR COMPUTING THE DORA METRICS

In this section, we describe our framework for automatically measuring our operationalization of the DORA metrics for projects that use version control or project hosting software (e.g., git and GitHub). Although our implementation is specific to git and GitHub, a similar approach can be used for other environments. We describe our implementation of this framework, the core terminology we use, and the algorithms we rely on for computing the metrics.

A. Terminology and Definitions

Deployment. *A deployment is defined as a git tag which follows a semantic versioning format and comes in a reasonable order.*

By "reasonable", we mean that two consecutive git tags must be created in order, such that the patch version is incremented, or such that the minor or major version is incremented, while all less significant versions are 0. However, multiple "branches" of tags are allowed, such that two consecutive tags can be considered valid if they are ordered in relation to a non-consecutive series of tags that appear nearby in the ordering, even if their direct neighbours contain significantly different major versions. In practice, this means that a project can simultaneously release patches to a 16.x.x branch, as well as to an 18.x.x branch, and that these deployments will be correctly identified. The following sequence would be considered a valid ordering: v16.3.2, v16.3.3, v18.0.1, v16.4.0, v18.1.0, v16.4.1; while the following would not: v16.3.2, v18.3.1, v16.3.4, v18.3.2, v16.3.3. In the latter case, the v16.3.3 tag is considered out of order as it comes after the v16.3.4 tag. In reality, this tag was likely a mistake, and was intended to be v18.3.3, but since this cannot be determined automatically, the tag is not considered to be a deployment.

While the definition of a deployment is the tag itself, it is also important to consider the unique git commit which the tag is applied to as the deployment, as this commit is synonymous with the tag. This allows us to consider deployments as nodes of the git commit graph, enabling concepts like *containment*, which we describe later.

Our decision to use git tags to represent deployments was made to serve the broadest range of projects possible, including projects without clearly defined cloud environments. Tagging software releases with versioned tags is a common industry practice [16], and as such we propose it as a reasonable restriction. (Indeed many projects which *do* have clearly defined cloud environments trigger updates to said environments upon the creation of new tags [16].) In a literature review of sixteen articles on the DORA metrics, Sallin et al. [8] describe *for the purposes of automated measurement [of the DORA metrics], a deployment is defined as a new release*, while Rios et al. [16] describe tags as being synonymous with releases for the purpose of git workflows.

While we use a broad definition of a deployment, we do acknowledge that a more restrictive definition of a deployment may be useful for specific studies, such as limiting deployments to tags which increment the major version, or excluding tags which have been marked by GitHub as a "prerelease", or which were created by bots.

Failure. *A failure is defined as a GitHub issue which is in the "completed" state, is labelled with a string indicating that the issue contains a bug, is not labelled with a string indicating that the issue is something other than a fixed bug, and is fixed by a commit.*

Labels which indicate that a GitHub issue is a bug can take different forms from project to project, but will generally contain the word "bug", either at the beginning of the label, or prefixed with a hyphen, space, or colon. Likely examples are "bug", "type:bug", "kind-bug", and "confirmed bug". Labels which indicate that an issue is not a fixed bug contain keywords such as "invalid", "wontfix", or "feature". This extra step is required as some issue reports will be labelled as "bug" by the reporter, but eventually tagged as "wontfix" by a contributor.

We acknowledge that other interpretations of failures exist, and that our definition is quite inclusive. We discuss the implications of this decision further in Section V. Indeed, Sallin et al. describe failures as requiring "human interpretation" and being "difficult to automate" [8]. The terminology *fixed-failure* may be more effective and descriptive for our use-case, but we have chosen *failure* in order to stay consistent with Accelerate [6] and the State of DevOps Report [1].

Containment. *A git commit is contained by a deployment if it is reachable on the git commit graph by the deployment, but not reachable by any previous deployments.*

Since deployments exist on the commit graph through the unique commit associated with the deployment's tag, the commit graph can be used to determine the relationship between specific commits and specific deployments. By traversing the commit graph backwards from the deployment, a set can be formed of all of the commits reachable from the deployment, representing all of the commits which are chronologically previous to the deployment, and on the same branch, or a branch which eventually merges. Then by repeating this process for the previous deployment, the difference between these two sets represents all of the new commits which are logically included by the deployment being examined. Our approach to containment mirrors that of the *range strategy* proposed by Pinto et al. [17]

Fixed. *A failure is fixed by a commit if the commit is included directly by GitHub's "closed" event for the failure, or if it is included in a pull request which links to the failure.*

Contributors can close GitHub issues either automatically or manually. To automatically close an issue, contributors can create a commit message which references the issue they desire to close by number, prefixed with a "#" symbol and the word "closes". An example commit message may be "Fixes the null exception, closes #132". This commit will direct GitHub to automatically close issue 132, and as such the commit is said to "fix" issue 132. Contributors can also manually close an issue when they decide that it has been addressed. In the event that this issue was fixed by new commits, those commits usually come in the form of a pull request which will reference the issue being fixed. In this case, the merge commit of the pull request being merged is said to "fix" the issue.

Some projects do not follow this documentation strategy, and commit bug fixes without appropriate documentation, or provide documentation which GitHub is unable to detect, and therefore is unable to use to create a link between the commit and the issue. These projects are not suitable for automatically calculating stability metrics.

Induced. *Induced. A failure is induced by a commit if the commit contains a bug that causes the failure.*

Sliwerski et al. defined the term fix-inducing commits as commits which cause problems and induce a bug-fix in the future [12]. The term is synonymous with a commit which causes a bug, and in this paper is extended to mean a commit which causes a failure.

B. Computing the DORA Metrics

Using the terminology defined previously, we describe how we compute the DORA metrics using four standardized algorithms. Our framework was designed with GitHub mining in mind, but can easily be modified to target non-GitHub or even non-git projects. The throughput metrics of *deployment frequency* and *lead time for changes* are mined only from git. Other version control software such as subversion could be mined for these metrics, assuming that they have equivalent concepts for tags and commits and share a similar internal structure to git repositories. The stability metrics of *mean time to recover* and *change failure rate* are mined using GitHub data, but only as a source of failures and bug-fixing commits, which we chose to derive from GitHub issues. These could be mined from other remote providers such as GitLab, or from a CVE database, so long as a list of failures and bug-fixing commits can be created. Indeed, recent versions of GitLab provide a number of calculated metrics to support usage analysis, and these metrics include the DORA metrics [18].

In our implementation, we use the SZZ algorithm [12] (we use the recommended improvements from SZZ Unleashed [19] which report better recall and precision) to create a list of bug-inducing commits from the list of bug-fixing commits already mined for *mean time to recover*, but this step and therefore the SZZ algorithm can be replaced by a different method to generate a list of bug-inducing commits.

We also note that in our algorithm for *deployment frequency*, we define an arbitrary time range t_0 to t_n. This range can be any time period being studied, or can be adjusted to a standardized size, such as a month or a year. In our results, we use the range of one year.

Algorithm 1 Deployment Frequency

```
1: Create the set $D$ of deployments within the time period $t_0$ to $t_n$
2: Calculate *Deployment Frequency* as $\frac{|D|}{t_n - t_0}$
```

Algorithm 2 Lead Time for Changes

```
1: Create the set $C$ of commits and the set $D$ of deployments.
2: Create the set $P$ of all pairs $(c, d)$ where $c \in C$ and $d \in D$ such that $c$ is *contained* by $d$.
3: Create the set $P_T$ of pairs $(t_d, t_c)$ for each $(c, d) \in P$ where $t_d$ is the deployment time of $d$ and $t_c$ is the commit time of $c$.
4: Create the set $T$ of times $t$ for each $(t_d, t_c) \in P_T$ where $t = t_d - t_c$
5: Calculate *Lead Time for Changes* as $\frac{1}{|T|} \sum_{t \in T} t$
```

Algorithm 3 Mean Time to Recover

```
1: Create the set $C$ of commits and the set $F$ of failures.
2: Create the set $P$ of all pairs $(c, f)$ where $c \in C$ and $f \in F$ such that $c$ *fixes* $f$.
3: Create the set $D$ of all pairs $(d, f)$ where $d$ is a deployment which *contains* $c$ and $(c, f) \in P$
4: Create the set $P_T$ of pairs $(t_d, t_f)$ for each $(d, f) \in D$ where $t_d$ is the deployment time of $d$ and $t_f$ is the time when $f$ is reported.
5: Create the set $T$ of times $t$ for each $(t_d, t_f) \in P_T$ where $t = t_d - t_f$
6: Calculate *Mean Time to Recover* as $\frac{1}{|T|} \sum_{t \in T} t$
```

Algorithm 4 Change Failure Rate

```
1: Create the set $C$ of commits, the set $F$ of failures, and the set $D$ of deployments.
2: Create the set $P$ of all pairs $(c, f)$ where $c \in C$ and $f \in F$ such that $c$ *induces* $f$
3: Create the set $D_f$ of all pairs $(d, f)$ where $d$ is a deployment which *contains* $c$ and $(c, f) \in P$
4: Create the set $D_u$ of all unique deployments $d$ where $(d, f) \in D_f$ and $f$ is any arbitrary failure
5: Calculate *Change Failure Rate* as $\frac{|D_u|}{|D|}$
```

IV. Automatically Measuring the DORA Metrics of Open Source Projects

To demonstrate the usefulness of our framework, we selected 304 open source projects to measure and compare the DORA metrics for. We first selected the top 1,000 most starred GitHub projects in the programming languages C, C++, Java, JavaScript, Typescript, Python, Go, Rust, PHP, and Swift. Then, in order to ensure the projects met our basic requirements of tagging version numbers and labelling issues as bugs, we filtered the list to include only projects which contained at least 50 recognizable deployments, and 100 recognizable failures. For each project in the final sample, we used the framework described above to calculate the four DORA metrics. We considered these metrics for each project year over year, since they were created up until November 2022.

First, we describe the results of automatically measuring the four DORA metrics on a per-year basis for the projects in our sample from 2013 to 2022. Then, using the comparison benchmarks set out by the 2021 State of DevOps report, we compare these results to the performance clusters the report also defined. We use 2021 for comparison as the DORA metrics are "lagging" indicators, and our 2022 data is only complete through November.

A. The DORA Metrics for 304 Popular Open Source Projects

Looking at the DORA metrics calculated for each year for our sample projects overall, we found that the DevOps performance of our sample has changed over time. Notably, we found that *deployment frequency* has shown a consistent increase, while *mean time to recover* doubled between 2016 and 2022 (see Fig. 1).

Considering the metrics for 2021 for our sample projects, we found that the median *deployment frequency* was 24.5 deployments per year. The median *lead time for changes* for our sample was 24.9 days, and the median *mean time to recover* was 95.2 days. The median *change failure rate* was 60.4%. In Table II, we show the spread of each metric for 2021, and found a significant variance between the top 25% and bottom 25% of projects for every metric.

B. Project storytelling using the DORA Metrics

In addition to looking at the aggregated DORA metrics across our sample of open source projects, we explored three

TABLE II
2021 PERFORMANCE SPREAD PER METRIC[1]

Percentile	Deployment Frequency [2]	Lead Time for Changes [3]	Mean Time to Recover [4]	Change Failure Rate
Top 25%	116.2	5.1	26.5	19.8%
Middle 50%	26.2	26.8	97.9	59.4%
Bottom 25%	6.6	140.6	370.0	90.6%

[1] Values are calculated as the mean of all projects within the percentile.
[2] *Deployment Frequency* represents median deployments per year.
[3] *Lead time for changes* represents median days between commit and deployment.
[4] *Mean Time to recover* represents median days between failure and recovery.

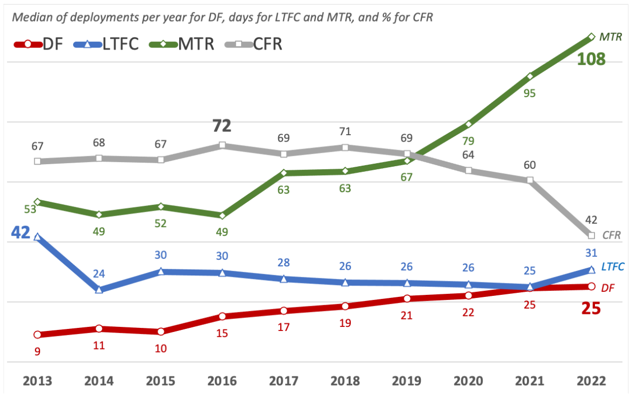


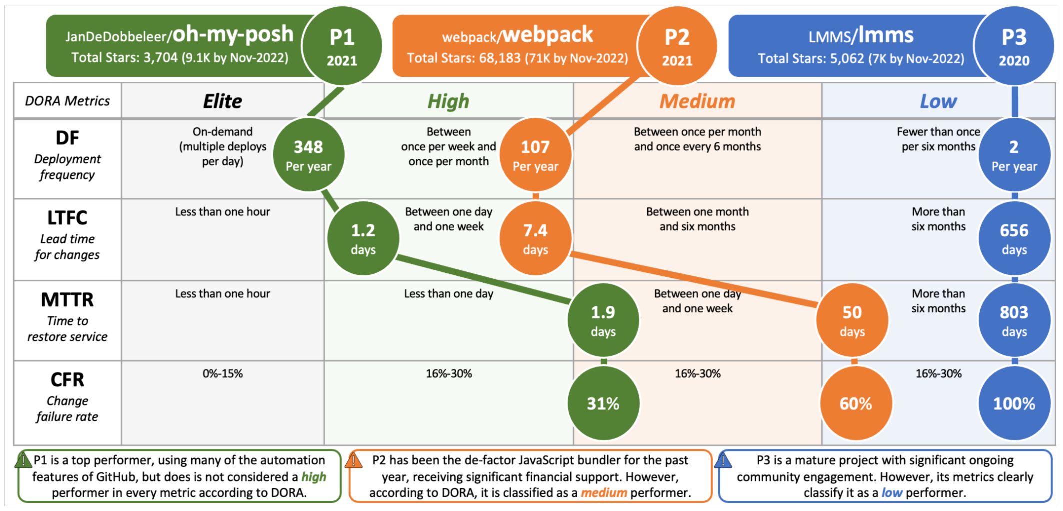
Fig. 1. Performance distribution over the period of 2013 to 2022. Datapoints are calculated as the median of all projects in the sample for the given year. *Deployment frequency* (DF) is measured as deployments per year, *lead time for changes* (LTFC) and *mean time to recover* (MTR) are measured in days, while *change failure rate* (CFR) is a percentage value.

specific projects: according to the DORA metrics, one of the top performing projects from our sample, one project with moderate DevOps performance but clear project success, and one of the worst performing projects from our sample. The DORA metrics for these three projects are shown in Fig. 2 and they are overlaid on the 2021 State of DevOps Report benchmarks. We briefly describe each of these projects and share what interesting trends and insights the DORA metrics can provide about them, while also highlighting some of the shortcomings of considering the DORA metrics in isolation.

The *JanDeDobbeleer/oh-my-posh* project (referred to as P1 below) makes effective use of DevOps practices and is one of the top performers in our study according to the four DORA metrics. P1 takes advantage of many automation features offered by GitHub, such as pull-request templates, repository bots, and continuous integration. Despite its high performance in our sample, it had a *mean time to recover* of 1.99 days in 2021 and would be considered only as a *medium* performer according to DevOps report benchmarks. P1 also had a *change failure rate* in 2021 of 31%, which falls slightly below even the lowest cluster defined by the State of DevOps Report. What the metrics do not tell us is that P1 receives limited funding, and is primarily maintained by a single developer, who has made 63% of all commits to the project. The efforts P1 has put into DevOps practices has a clear community benefit, as there are many examples of bug-reporters expressing gratitude for the speed and effectiveness of bug fixes in the P1 community. P1's excellent *mean time to recover* and *deployment frequency* show that the tool has excellent community support, and would be a good choice to use or sponsor.

The second project we explored is the *webpack/webpack* project (we refer to it as P2 below). P2 is a very successful project [20]. It is a static module bundler for JavaScript applications and is a core technology to the full-stack web development industry [21]. Similarly to P1, one engineer accounts for a significant percentage of the development of P2 but it does receive a significant amount of funding [22] and employs a development and community management team. This financial support indicates not only interest from the community (P2 has 71,230 stars as of November 2022), but that P2 is an important dependency for many development teams and companies. However, when compared to the clusters defined in the 2021 State of DevOps report, P2 would only be considered as a *medium* performer in terms of its DevOps practices, which does not capture the project's clear success. The State of DevOps Report and the Accelerate book both mention that the DORA metrics correlate with business outcomes, but this may be harder to show for open source projects. However, it may also be that the benchmarks for software projects in general do not work well as benchmarks for open source projects specifically.

The third project we selected, as a *low* performing example is *LMMS/lmms*, the Linux MultiMedia Studio, a music production software project (we refer to it to as P3 below). P3's first release was in 2005 and since then it has received consistent community engagement, with 7,036 stars as of November 2022. In terms of its DORA metrics, it is one of the lowest performers in our sample. Despite this, by analyzing other metrics, we can see that this project is still very active in its community. Figure 3 shows the DORA metrics, some additional development metrics, and overlays some keys events from 2014 until 2020. In 2016, we observe a drastic reduction of the software delivery and operational performance metrics (see Fig. 3-b). Looking further into its history, we notice that the main contributor and co-founder stopped contributing regularly in 2015. The commit frequency of the aforementioned contributor decreased from 206 commits per year from 2005-2015 to just 3 commits per year from 2016-2022. During 2015 (see Fig. 3-a), the project deployed 27 times, increasing from 10 deployments in 2014, but then proceeded to deploy 0 times in 2016. From 2017-2020, P3



JanDeDobbeleer/oh-my-posh
Total Stars: 3,704 (9.1K by Nov-2022)
P1 2021
webpack/webpack
Total Stars: 68,183 (71K by Nov-2022)
P2 2021
LMMS/lmms
Total Stars: 5,062 (7K by Nov-2022)
P3 2020
DORA Metrics
Elite
High
Medium
Low
DF Deployment frequency
On-demand (multiple deploys per day)
348 Per year
Between once per week and once per month
107 Per year
Between once per month and once every 6 months
Fewer than once per six months
2 Per year
LTFC Lead time for changes
Less than one hour
1.2 days
Between one day and one week
7.4 days
Between one month and six months
More than six months
656 days
MTTR Time to restore service
Less than one hour
Less than one day
1.9 days
Between one day and one week
50 days
More than six months
803 days
CFR Change failure rate
0%-15%
16%-30%
31%
16%-30%
60%
16%-30%
100%
P1 is a top performer, using many of the automation features of GitHub, but does is not considered a high performer in every metric according to DORA.
P2 has been the de-factor JavaScript bundler for the past year, receiving significant financial support. However, according to DORA, it is classified as a medium performer.
P3 is a mature project with significant ongoing community engagement. However, its metrics clearly classify it as a low performer.

Fig. 2. Example of three open source repositories and their classification in comparison to the benchmarks defined by the 2021 State of DevOps Report [1]. (P1) *oh-my-posh* as *High*, (P2) *webpack* as *Medium*, and (P3) *lmms* as an example of a *Low* software delivery and operational performance repository.

had a *deployment frequency* of 3 deployments per year, and has not deployed since.

Although the project's DevOps performance has declined and the team has lost an important contributor, community interest and engagement has not disappeared. The total number of stars has increased from 263 in 2014 to 5,052 in 2020 (see Fig. 3-c). Additionally, even when the project's DORA metrics were at their worst in 2016, development did not stop, with P3 gaining 526 *new issues opened*, 334 *pushes*, and 521 *commits* that same year from other contributors.

In summary, the three projects we explored in depth gives us important insights into what the DORA metrics can and cannot tell us. The metrics are a good starting point to get an overview of a project's software delivery performance, but further context is needed in order to gain a complete understanding of a project's health. However, we found that analyzing and visualizing the DORA metrics over time is a useful approach to understand the history of a project, especially when this time series data can be correlated with significant events, such as changes in team size or membership.

V. Discussion

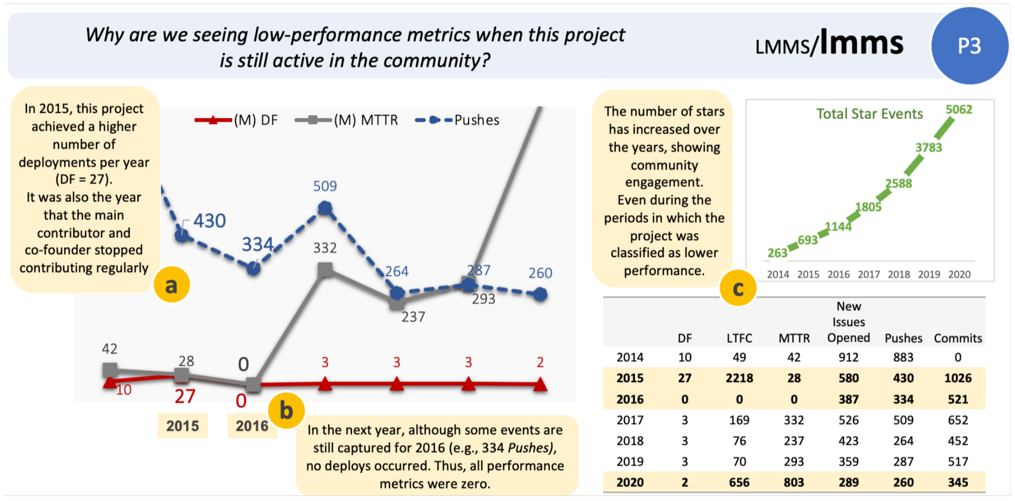
Our exploration into automatically measuring the DORA metrics of 304 open source projects yielded numerous points for discussion. We start by discussing some of the differences between telemetry and survey approaches to measuring the DORA metrics, then compare our data with the survey-based benchmarks set by the 2021 State of DevOps Report. From there we investigate how our framework reveals dependencies between each of the metrics, discuss some of our design choices in more depth, and finally mention areas for future research

A. Benefits of Telemetry Data

Sallin et al. mention in their work that the method used for measuring the DORA metrics introduces different strengths and weaknesses [8]. The State of DevOps Report methodology relies on survey data. The advantage of survey data is that it does not depend on specific DevOps tooling for the analysis. However, with surveys, subjective responses and respondent bias can be a concern, and indeed the most recent State of DevOps Report in 2022 mentions that the respondents to that survey may not have been as representative of projects that effectively use DevOps practices compared to prior years [7]. This difference in responses may explain why their results in 2022 vary from the trends in the previous years. Survey fatigue can also be a factor.

Using telemetry data in a standardized framework can create accountability and remove the bias which may come from subjective survey data. It allows studies to be reproducible, something the State of DevOps Reports have previously been criticized for [23]. Furthermore, terms such as "failure" can be defined formally (removing human varied interpretations), creating an even playing field for comparisons across multiple projects.

Using telemetry data also allows for the creation of time series comparisons, which in turn can support retroactive analysis of major events in a project's life-cycle. The effect of adding new technologies to a project's tool-chain, receiving additional funding, or restructuring the development team can be directly observed as changes (or the absence of changes) in a project's metrics. When examining the DORA metrics for *lmms* (see Figure 3-c), it was apparent that a significant event took place during 2015, despite traditional open-source metrics such as stars and pushes remaining unchanged (or improving). The DORA metrics may prove useful for retroactively analyzing other significant events in a project's life-cycle, such as the ones mentioned above. Research on this topic could be expanded to include comparisons and predictions. Models could be developed to help predict how a project's DORA metrics might change given certain decisions, based on how those same decisions affected the metrics of similar open



Why are we seeing low-performance metrics when this project is still active in the community?

In 2015, this project achieved a higher number of deployments per year (DF = 27). It was also the year that the main contributor and co-founder stopped contributing regularly

In the next year, although some events are still captured for 2016 (e.g., 334 *Pushes*), no deploys occurred. Thus, all performance metrics were zero.

The number of stars has increased over the years, showing community engagement. Even during the periods in which the project was classified as lower performance.

	DF	LTFC	MTTR	New Issues Opened	Pushes	Commits
2014	10	49	42	912	883	0
2015	**27**	**2218**	**28**	**580**	**430**	**1026**
2016	**0**	**0**	**0**	**387**	**334**	**521**
2017	3	169	332	526	509	652
2018	3	76	237	423	264	452
2019	3	70	293	359	287	517
2020	**2**	**656**	**803**	**289**	**260**	**345**

Fig. 3. Exploring *lmms* repository events and software delivery and operational performance metrics over the years. Three highlights observed: (a) the repository numbers when the main contributor and co-founder stopped contributing regularly in 2015. (b) 2016 as the first year observed where *deployment frequency* is high. (c) Latest metrics observed for this repository was in 2020, a *Low* performer but with active community engagement.

source projects.

B. Comparing the Automatically Measured DORA Metrics with the State of DevOps Report

In this section we compare our results to the 2021 State of DevOps Report [1] benchmarks. While the report provides these benchmarks for exactly this purpose, it is worth noting that they are derived from survey data, and it is unclear what methods survey respondents used to measure their metrics, (the surveys are not reproducible). As such these comparisons are only anecdotal, and show why a standardized framework is needed for accurate comparisons.

Considering the 2021 State of DevOps report [1] benchmark chart (see Table I), the median *deployment frequency* for our sample of open source projects in 2021 would be classified at the low end of the *high* performance cluster from the 2021 report, while the median *lead time for changes* would be classified between the *medium* and *high* performance clusters. However, both the median *mean time to recover* overall, and the mean of the top 25% of projects by *mean time to recover* (see Table II) would be classified between the *low* and *medium* clusters according to the State of Devops report. The median *change failure rate* was also significantly higher than the 15%-30% benchmark which the State of DevOps report indicates for even the *low* performance cluster.

When comparing the performance of individual metrics in 2021 to the benchmarks set out by the 2021 State of DevOps report, 75.6% of projects included in our study would be classified as *high* performers when considering only *deployment frequency*, while 78.2% would be considered *medium* performers when considering only *lead time for changes*. This indicates moderately high performance for throughput compared to the clusters defined by the 2021 report. However, we see a different picture for the stability metrics, as stability would be considered lower for our sample. 76.8% of the projects would be classified as *medium* performance when only considering *mean time to recover*, and 76.3% of the projects have a worse *change failure rate* than even the lowest data points considered by the 2021 State of DevOps report. We show a complete comparison of our sample to the 2021 benchmarks in Fig. 4.

This prioritisation of throughput over stability may come from the type of projects included in the study. While outages for software as a service (SaaS) products can have severe consequences, versioned products such as frameworks and libraries can have a culture of acceptance around bugs [24]. While a 25% *change failure rate* is classified as *low* performance by the DevOps report, developing a framework where three in every four releases have zero bugs is a significant achievement. Similarly, a week-long outage for a software as a service product is dramatic, while releasing bug fixes within a week of their corresponding bug reports is certainly within a reasonable time-frame for open source libraries [25]. However, we found that the bottom 25% of projects in our sample had a mean *lead time for changes* of 140.6 days in 2021 and a mean *mean time to recover* of 370 days (see Table II). This lead time indicates that many projects in our sample are regularly deploying commits which approach or exceed half a year in age, a practice which has clear security and productivity implications.

C. Dependencies Between Metrics

By formalizing the methods for measuring the DORA metrics, our framework reveals some interesting dependencies between them. We can see that *deployment frequency* directly influences all three of the other metrics. Although it is only one of a number of factors driving the other metrics, a high *deployment frequency* can allow projects to also have a low *lead time for changes*, low *mean time to recover*, and low *change failure rate*, while a low *deployment frequency* is either

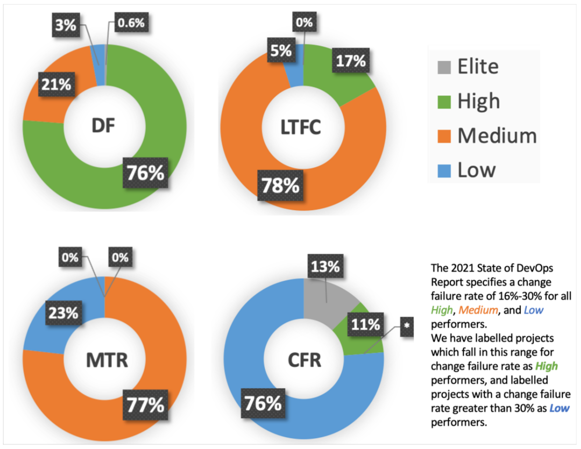


Fig. 4. Distribution of projects in our sample, when comparing each DORA metric from 2021 to the 2021 State of DevOps Report benchmark [1].

a sign of project stagnation, or a sign of missing DevOps practices (noting that some open source projects may employ DevOps practices using other tools outside of the normal GitHub workflow). According to the laws of evolution, most projects will innovate and change over time [26], and a low *deployment frequency* bottleneck may mean little innovation is happening. However, in the case of open source projects that are stable and still useful, it may be expected that *deployment frequency* decreases, even when the software is widely used and some community members continue to request changes.

Despite this one way dependency, *lead time for changes* is not directly correlated with *deployment frequency*, as a different project reporting the same *deployment frequency* of one deployment per year may still achieve a low *lead time for changes* if the commits being deployed were all committed in a short time period leading up to the yearly deployment. As such, *deployment frequency* and *lead time for changes* are both needed to capture a project's throughput performance, as each one independently does not capture enough context. Similarly, *mean time to recover* is influenced by both of the stability metrics.

Teams which regularly commit bug fixes within a couple days of their corresponding bug reports will still report a higher mean time to recover if there are bottlenecks in code review (a high *lead time for changes*), or if the team does not deploy a new version for a significant time period (a low *deployment frequency*).

D. Effectively Representing Failures

Our inclusive definition of a *failure* allows our framework to include a broad range of projects, but may produce worse stability metrics than a project might expect. The State of DevOps Report describes a failure to be any event which leads to degraded service or requires remediation of any kind, including a "hotfix" or "patch". Our definition of a failure is inclusive of all verified bugs which are eventually fixed, regardless of criticality. While we decide to include all fixed bugs as failures, we recommend that projects which record bug criticality should use said criticality to better inform their stability metrics. For example, if a project considers three levels of bug criticality, a separate stability metric could be calculated for each level, only considering failures of said level.

Existing automated solutions consider failures to be incidents reported by automated infrastructure monitoring systems. For projects without a clear cloud deployment (to a server or similar infrastructure), this will not work. As such we consider reported bugs to be the most reliable source of a failure. This definition may fail to recognize bugs which nobody finds or reports, or bugs which are never fixed. Still, we consider it reasonable to exclude such examples, as DORA themselves indicate that a failure requires a patch or revision. While we identify bugs using GitHub, we designed our framework such that bugs from other sources, such as SVE lists, could easily be considered, and recommend future work investigate this.

E. Research Directions

By formally operationalizing each of the DORA metrics, we present a framework which can be used to study the DORA metrics of individual projects, or compare the metrics of varying samples of projects. We chose to study a limited sample of previously neglected open source projects to show the applicability of our framework, but call for future research utilizing our framework to automatically measure the DORA metrics for other lists of projects. The Linux Foundation's critical open source software projects list [27] would be an interesting sample to investigate, as would older established repositories, such as the Unix Kernel [28], to see how the DevOps movement has affected these projects.

We see particular value in studying possible correlation between the DORA metrics and other project quantities. The State of DevOps Report [1] shows a strong correlation between the metrics and business success, and these claims can be replicated using a common framework for DORA metric measurement. Specific quantities such as project funding, team size, GitHub Stars, Twitter mentions, and Google search hits could be investigated in relation to a projects DORA metrics. For open source projects, research could also be done to investigate if high DevOps performance correlates with project survival [29] or developer satisfaction [30].

In addition to providing a standardized and replicable method to measure the DORA metrics, our framework opens the door for retroactive analysis of large events in a project or organization's history. We call for future research investigating how changing leadership, changes to funding, or changes to team structure effect a projects DORA metrics in the short and long term future. Similarly, time series data allows studies to investigate the trending direction of specific projects, or subsets of the software industry. Studies could for example investigate the trending direction of DORA metrics for projects using Rust, compared to those using C++. These comparisons are made possible by our standardized, project-agnostic framework.

VI. Threats to Validity

As our project is an initial investigation into automatically measuring the DORA metrics, there are several threats to our study, many of which point to future work as discussed earlier. In terms of **construct validity**, the way we measure the DORA metrics is a possible threat. We are fairly confident with how we measure the throughput metrics, but are less confident with how we measure the stability metrics. Other researchers have also discussed challenges measuring *change failure rate* in particular and likewise find that how this metric is defined by DORA is not clear [8]. We also only measure failures which are fixed, which is another threat. In calculating *change failure rate*, we used the SZZ algorithm to map bug-fixing commits to bug-inducing commits. While we applied recommended improvements, such as excluding whitespace and comments from diff-reports, the SZZ algorithm is known to show bias and produce inaccurate results [13].

With respect to **internal validity**, we compared the DORA metrics for the open source projects in our sample to the 2021 State of DevOps Report. However, we noted that this comparison may not be entirely accurate given that the benchmarks use survey data rather than telemetry data. Future work should explore this threat, along with the validity of the State of DevOps Report benchmarks.

In terms of **external validity**, we recognize our sample is small (and not representative of all open source projects), future work may consider larger and different samples as we discussed above. In selecting the 304 repositories used in the study, we had to strike a balance between filtering repositories which did not adequately use GitHub and over-biasing towards projects embracing DevOps. We chose to exclude projects with fewer than 50 recognizable deployments, and fewer than 100 recognizable failures. This ensured that each project we included could produce a reasonable quantity of telemetry data, but we may have excluded projects which were in fact mature projects using GitHub, and simply had extremely low rates of bugs or versions. This selection process only showcased a subset of the open source community.

We make all of our scripts, tools, data and findings available as part of our replication package [3], as well as an interactive dashboard to visualize the metrics of specific projects in detail.

VII. Concluding Remarks

The DORA metrics have seen explosive adoption across the industry for measuring DevOps performance and enabling organizations to compare their metrics with industry benchmarks. These metrics have been shown to correlate with business success and developer satisfaction [1], however existing methods to compute them are highly restrictive, requiring specific types of cloud infrastructure, project structure, or software development life cycle approaches. As such many types of projects, such as libraries, frameworks, and programming languages cannot currently measure their metrics [5].

[3]Replication Package: https://zenodo.org/record/7874612

Our study proposes a novel framework for automatically measuring the DORA metrics in a project-agnostic way, using telemetry data mined from a projects git and Github repositories. We demonstrate the general applicability of our framework by automatically measuring the DORA metrics of 304 open source projects on GitHub. We showed how our reproducible approach allowed us to uncover interesting insights into three specific projects, as well as reveal interesting trends across the entire sample.

We find that our framework provides a standard way to automatically measure and compare the DORA metrics of a broad range of software projects, but we recommend augmentations to our approach based on the availability of data such as bug criticality. We provide a replication package, and we hope that these contributions and our research encourages further research and strategies to improve DevOps effectiveness of projects across the software industry.

VIII. Acknowledgment

We'd like to thank Daniel German, Joseph Howie, Derek Robinson, Kezia Davathasan, and Leslie Wilkes for their support and assistance in this study.

References

[1] Google, "DevOps research and assessment. 2022 state of devops report," Dec 2022. [Online]. Available: https://services.google.com/fh/files/misc/state-of-devops-2021.pdf

[2] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "Devops," *Ieee Software*, vol. 33, no. 3, pp. 94–100, 2016.

[3] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *2009 6th IEEE International Working Conference on Mining Software Repositories*, 2009, pp. 1–10.

[4] Humanitec, "DevOps setups: A benchmarking study 2021," Dec 2022. [Online]. Available: https://f.hubspotusercontent40.net/hubfs/5890440/Reports/DevOps%20Setups_%20A%20Benchmarking%20Study%202021_v1.1.pdf?utm_medium=email&_hsmi=160478735&utm_content=160478735&utm_source=hs_automation

[5] Atlassian, "2020 devops trends survey," Atlassian, Tech. Rep., 2020. [Online]. Available: https://www.atlassian.com/whitepapers/devops-survey-2020

[6] N. Forsgren, J. Humble, and G. Kim, *Accelerate: The science of lean software and devops: Building and scaling high performing technology organizations*. IT Revolution, 2018.

[7] Google, "DevOps research and assessment. 2022 state of devops report," Dec 2022. [Online]. Available: https://services.google.com/fh/files/misc/2022_state_of_devops_report.pdf

[8] M. Sallin, M. Kropp, C. Anslow, J. W. Quilty, and A. Meier, "Measuring software delivery performance using the four key metrics of devops," in *International Conference on Agile Software Development*. Springer, Cham, 2021, pp. 103–119.

[9] J. A. Hagenaars and A. L. McCutcheon, *Applied latent class analysis*. Cambridge University Press, 2002.

[10] S. I. Vrieze, "Model selection and psychological theory: a discussion of the differences between the akaike information criterion (aic) and the bayesian information criterion (bic)." *Psychological methods*, vol. 17, no. 2, p. 228, 2012.

[11] J. Courtiel, P. Dorbec, and R. Lecoq, "Theoretical analysis of git bisect," in *Latin American Symposium on Theoretical Informatics*. Springer, 2022, pp. 157–171.

[12] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[13] S. Herbold, A. Trautsch, F. Trautsch, and B. Ledel, "Problems with szz and features: An empirical study of the state of practice of defect prediction data collection," *Empirical Software Engineering*, vol. 27, no. 2, pp. 1–49, 2022.

[14] S. Kim, T. Zimmermann, K. Pan, E. James Jr *et al.*, "Automatic identification of bug-introducing changes," in *21st IEEE/ACM international conference on automated software engineering (ASE'06)*. IEEE, 2006, pp. 81–90.

[15] H. Borges, M. T. Valente, A. Hora, and J. Coelho, "On the popularity of github applications: A preliminary note," *arXiv preprint arXiv:1507.00604*, 2015.

[16] J. C. C. Rios, S. M. Embury, and S. Eraslan, "A unifying framework for the systematic analysis of git workflows," *Information and Software Technology*, vol. 145, p. 106811, 2022.

[17] F. C. d. R. Pinto and L. G. P. Murta, "On the assignment of commits to releases," *Empirical Software Engineering*, vol. 28, no. 2, p. 32, 2023.

[18] "Gitlab dora metrics," Jan 2023. [Online]. Available: https://docs.gitlab.com/ee/user/analytics/dora_metrics.html

[19] M. Borg, O. Svensson, K. Berg, and D. Hansson, "Szz unleashed: an open implementation of the szz algorithm-featuring example usage in a study of just-in-time bug prediction for the jenkins project," in *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, 2019, pp. 7–12.

[20] S. Laurila, "Comparison of javascript bundlers," 2020.

[21] F. Zammetti, *Modern Full-Stack Development: Using TypeScript, React, Node. js, Webpack, and Docker*. Springer, 2020.

[22] "Open collective for webpack," Jan 2023. [Online]. Available: https://opencollective.com/webpack

[23] K. Lee, "A review of accelerate: The science of lean software and devops," 2022, accessed: 2023-04-27. [Online]. Available: https://keunwoo.com/notes/accelerate-devops/

[24] L. Zhao and S. Elbaum, "Quality assurance under the open source development model," *Journal of Systems and Software*, vol. 66, no. 1, pp. 65–75, 2003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016412120200064X

[25] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 361–370. [Online]. Available: https://doi.org/10.1145/1134285.1134336

[26] M. M. Lehman, "Laws of software evolution revisited," in *European Workshop on Software Process Technology*. Springer, 1996, pp. 108–124.

[27] "Securing critical projects workgroup: List of projects identified as critical," Jan 2023. [Online]. Available: https://docs.google.com/spreadsheets/d/1ONZ4qeMq8xmeCHX03lIgIYE4MEXVfVL6oj05lbuXTDM/edit#gid=571311621

[28] D. Spinellis, "A repository with 44 years of unix evolution," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 462–465.

[29] A. Ait, J. L. C. Izquierdo, and J. Cabot, "An empirical study on the survival rate of github projects," *International Conference on Mining Software Repositories (MSR)*, pp. 365–375, 2022.

[30] N. Forsgren, M.-A. Storey, C. Maddila, T. Zimmermann, B. Houck, and J. Butler, "The space of developer productivity: There's more to it than you think." *Queue*, vol. 19, no. 1, pp. 20–48, 2021.