Snapshot Metrics Are Not Enough: Analyzing Software Repositories with Longitudinal Metrics


* Loyola University Chicago, Computer Science Department
† Purdue University, Electrical and Computer Engineering Department
‡ University of California at Davis
§ Duke University

Abstract—Software metrics capture information about software development processes and products. These metrics support decision-making, e.g., in team management or dependency selection. However, existing metrics tools measure only a snapshot of a software project. Little attention has been given to enabling engineers to reason about metric trends over time—longitudinal metrics that give insight about process, not just product. In this work, we present PRIME (PRocess MEtrics), a tool for computing and visualizing process metrics. The currently-supported metrics include productivity, issue density, issue spoilage, and bus factor. We illustrate the value of longitudinal data and conclude with a research agenda. The tool’s demo video can be watched at https://youtu.be/YigEHy3. The source code can be found at github.com/SoftwareSystemsLaboratory/prime.

I. INTRODUCTION

An effective software engineering process is correlated with high software quality [1]. Measurements of software processes give engineers insight into software quality [2]. Software metrics characterize the software engineering process (e.g., time to fix a defect) and the engineered product (e.g., cyclomatic complexity). Using software metrics, engineers and managers improve products and assess the risks of external software dependencies.

In our survey of existing tools for software metrics, we found that typical metrics tools provide aggregate metrics, or ‘snapshot metrics,’ rather than longitudinal metrics (§II). They capture the current state of the software product and perhaps some process metrics (e.g., age of current issues). While a snapshot can be useful—for example, it can quickly reveal if a project has no test suite—it does not provide a full picture of the longitudinal evolution of a software project. We conjecture that engineers will make different decisions when presented with snapshot metrics compared to considering the trends in a project’s metrics over time (§I).

To evaluate a development process, one needs to measure the history of the code. The classic reference on software metrics [2] establishes that measurement needs to be related to a time range and scale for a meaningful longitudinal assessment of software quality [2]. Tools that measure quality need to calculate both direct measurements and derived calculations at consistent intervals to evaluate the process properly. Historical trends can be visualized and utilized to quantify software engineering decisions by collecting and reporting process metrics at consistent intervals.

To support our investigation of this research question, we present PRIME [3] (PRocess MEtrics): an open-source tool that enables engineers and researchers to analyze software projects with longitudinal metrics. PRIME uses a modular Extract-Transform-Load pipeline architecture for ease of adoption and extension. PRIME currently supports the following metrics: code size, productivity, bus factor, issue count, issue spoilage, and issue density.

We close by proposing three studies facilitated by PRIME: (1) exploring engineers’ use of longitudinal metrics when assessing their products; (2) exploring their use of longitudinal metrics during dependency selection; and (3) analyzing the software supply chain to identify potential weak links.

II. BACKGROUND AND RELATED WORK

Process metrics are critical for improving software quality as agile repositories may eventually become more established and require regular maintenance. Although numerous efforts have focused on mining open-source repositories, the current support for process metrics—and visualizing them longitudinally—is mixed. In our survey of related efforts, we encountered various tool types, including scorecards, frameworks, dashboards, and platform monitors.

Scorecards assign a risk score for open source projects to assess security risks and project health. The OSSF/SCORECARD [4] However, they are computed as a snapshot metric cannot easily express longitudinal effects.

Frameworks simplify the mining software repository (MSR) development process. These are typically domain system languages (DSL) and libraries that researchers and engineers integrate into their tools. ISHEPARD/PYDRILLER [5] and the BOA [6] DSL meet this criteria. These frameworks allow for MSR tools to be developed for the analysis of VCS, but they are not MSR tools.

Dashboards are built into online VCS platforms designed to provide visualizations of repository and issue tracker trends. GitHub Insights [7] and GitLab Insights [8] provide longitudinal metrics for hosted projects. However, these tools provide limited insights when it comes to process metrics.

Aside from dashboards, these tools compute process metrics as snapshots and do not make longitudinal and trends visualization easy for users.

### III. Architecture

**PRIME** is architected as an Extract, Transform, Load (ETL) pipeline that utilizes modular components to compute and visualize metrics, as shown in Figure 1.

The ETL phases of the pipeline are each module or collection of modules. In addition, the extraction and transformation stages of the pipeline store data in text-encoded JSON files. By storing measurements in a file rather than in memory during pipeline execution, **PRIME** can be integrated with existing tools and pipelines.

**PRIME** extracts base measurements from a project’s version control system (VCS) and issue tracker during the API Phase. Here, using CLOC [12] and SLOCCount [13], **PRIME** measures each commit of a repository and measures the size of the repository in lines of code (LOC), thousands of lines of code (KLOC), and the size difference between each sequential commit as the delta thousands of lines of code (DKLOC). **PRIME** also extracts issue report metadata by utilizing the REST API of a repository’s host issue tracker.

**PRIME** transforms the extracted base measurements into derived metrics during its Metrics Phase. At the moment, **PRIME** can compute the following metrics: issue spoilage, issue/defect density, productivity, and bus factor. Each metric module takes in a text-encoded JSON file containing both or either the commits or issue base measurements.

After both the API and Metrics phases, data is loaded into either text-encoded JSON files or visualized with MatPlotLib [14] in the Output Phase. **PRIME** can export the JSON and visualization files to integrate with other pipelines. Additionally, the visualizations can be customized using style sheets and parameters for the major components, thereby allowing engineering teams to implement style standards for their visualizations.

This architecture allows engineers to download and install individual **PRIME** modules to measure specific attributes of their software without waiting for measurements of unwanted attributes. Furthermore, each phase of the pipeline is configurable, reducing the time engineering teams need to post-process the data to match their specific needs.

### IV. Metrics Implemented

**PRIME** computes two types of software metrics: (1) **Direct metrics**, which are measurements of internal attributes of the process, and (2) **derived metrics**, which are computed metrics from two or more direct metrics.

#### A. Direct Metrics

Direct metrics are measurements of a particular attribute of the process involving no other attribute. These measurements are the foundation for the more complex metrics that **PRIME** computes.

1. **Code Size**: **PRIME** measures the size of a repository in terms of the number of source lines of code normalized by 1000 reported as KLOC. Changes in the KLOC (DKLOC) show the growth (or shrinkage) of a repository over time.

2. **Developer Count**: **PRIME** measures this metric as the number of unique developers who contribute code to a repository within a time interval. By measuring developer count, engineering teams can determine the amount of developer support in contributing new code, maintaining existing code, and resolving bugs.

3. **Issue Count**: **PRIME** measures this metric as the count of the number of open and closed issues reported in an issue tracker, including feature requests, tasks, and bug reports, in addition to potential and confirmed defects. If an online VCS has an issue tracker, this metric also reports the count of open and closed pull requests.

#### B. Derived Metrics

Derived metrics are useful for analyzing the interactions between direct metrics. **PRIME** computes derived metrics to analyze and subsequently visualize changes in the development process of a software product.

1. **Issue Density**: This metric tracks a project’s total number of issues normalized by project size. Because we are interested in open-source repositories on GitHub, we use the more general issue density rather than defect density, which refers only to the ratio of bug count to repository size. A high issue density, regardless of confirmed defects, could signify an unhealthy repository. For example, if there are many feature requests that are never acted upon, then the development team is not implementing the features that users want. This would be a possible warning sign for poor customer support and, eventually, would lead to low customer or user satisfaction.

2. **Issue Spoilage**: Issue spoilage is the weighted average age of unresolved issues at a given time in the project timeline. With further analysis, this metric calculates the age of issues with respect to the project timeline to measure how quickly a project’s team resolves issues. Issue spoilage can serve as a gauge of customer support and the efficiency of software teams in resolving issues. For instance, if issue spoilage increases in...
Fig. 2: This figure shows the PRIME tool’s output for each supported longitudinal derived process metric applied to several sample projects. In the first pair 2a-2b, 2a sees a significant increase in issue density over its history with some recent improvement. In contrast, 2b saw a similar trend with dramatic improvement in issue density. The second pair 2c-2d shows two projects with contrasting trends in resolving issues. The third pair 2e-2f shows two projects with contrasting productivity trends. The first had a high level of productivity earlier in its history, while the second is showcases more recent evidence of high productivity. The fourth pair 2g-2h shows two projects with contrasting bus factor. The bus factor metric is binned at 30 days to represent the number of core contributors per month.

3. Productivity: Productivity measures the rate at which a development team adds KLOC within a time interval [2]. Healthy repositories will typically have high productivity. However, low productivity is not always a sign of a lack of productiveness, as when efficient development teams are refactoring code KLOC may not change significantly.

4. Bus Factor: Bus factor [16] is the number of developers on a project team who would have to be hit by a bus to cause the project to fail. This metric measures the employee turnover risk of a project. However, as our work focuses on open-source projects, we propose that this is a metric of the development community’s interest. As interest in a project fluctuates, we propose that a longitudinal analysis of this metric is key to assessing interest.

V. Demonstration

Figure 2 shows all four process metrics for several repositories over their entire project history. We chose projects from the REPOREAPERS/REAPER data set [17] in pairs that showed contrasting trends in their process metrics to demonstrate possible insights from longitudinal analysis. We have organized this figure to demonstrate the potential for comparative
analysis of process effectiveness, even among projects that have a good score using existing scorecard apps. The addition of process metrics clearly demonstrates that all of these otherwise good projects may benefit from further examining their development process. This examination is especially prudent when it comes to addressing issues (issue spoilage), managing development while addressing issues (issue density), ensuring appropriate resources (bus factor), or managing group priorities to avoid team burnout (productivity).

VI. PLANNED STUDIES

We intend to expand on the ideas presented in this paper through a few planned studies incorporating PRIME.

In the first study, we pose the research question: How do engineers use longitudinal process metrics during their development process? We hypothesize that basic metrics are used in many open-source projects today, but the use of longitudinal metrics, particularly process metrics, is limited. To perform this study, we will measure the number of process metrics utilized and survey open-source developers on established processes.

In a second study, we pose the research question: Do longitudinal metrics contribute to selecting dependencies in software composition? Based on our survey of tools, we hypothesize that engineers take little consideration of derived longitudinal process metrics but will consider direct longitudinal process metrics as those are more prevalent when selecting dependencies for software development. To perform this study, we intend to survey the current state of software metrics tooling, and survey open-source engineers about their utilization of longitudinal direct and derived process metrics for dependency selection.

In our third study, we hypothesize that many open-source projects today, but the use of longitudinal metrics, particularly process metrics, is limited. To perform this study, we will examine the number of process metrics utilized and survey open-source developers on established projects about why and how they use these metrics in their development process.

We intend to expand on the ideas presented in this paper through a few planned studies incorporating PRIME. Future studies will build on this foundation to study the usage of longitudinal metrics in practice, longitudinal metrics in selecting dependencies, and the software supply chain.

REFERENCES

count/