Pre-proceedings of

BENEVOL 2025

The 24th Belgium-Netherlands Software Evolution Workshop

17–18 November 2025

Enschede, The Netherlands

UNIVERSITY OF TWENTE.

Editors: Vadim Zaytsev, Fernando Castor.

Preface

The goal of BENEVOL has always been to bring together researchers who are working in the field of software evolution and maintenance. In addition to flagship venues like ICSME and SANER and other publication-oriented venues like ICPC, MSR, SCAM or SAC, the workshop nature of BENEVOL offers an informal forum to meet and discuss new ideas, relevant problems, and fresh research results.

This year we have welcomed submissions in the following four categories:

Cutting-edge Original Research on Evolution (CORE)

Full-length technical papers presenting original research which has been already empirically or formally validated. We also welcome papers in the early stages of their development that would benefit from feedback from the community.

Disruptive Ideas and Visionary Explorations (DIVE)

Position papers presenting new and potentially controversial software maintenance and evolution perspectives. Such papers may not have an evaluation, though illustrative examples and cases are welcome.

Reproduced, Examined, or Analysed Papers (REAP)

The growing complexity and size of software systems combined with the rise of machine learning for maintenance and evolution tasks makes replicability challenging. We invite replication efforts of existing papers, describing successes (replicated), difficulties (invalidated), or refinements (of any kind). This category extends to new negative results, which are key to narrowing down hypotheses.

Summary of Highlights and Outstanding Work (SHOW)

Presentation abstracts, progress reports, research results without inclusion in the proceedings. SHOW papers cover research that has already been accepted for publication, is being reviewed to be published, or is almost ready to be submitted to a conference or a journal.

All papers were submitted under a single-blind no-rebuttal reviewing model. SHOW papers were reviewed for thematic relevance, others were distributed among the reviewers in a way that each CORE, DIVE and REAP paper would get at least three reviews, assessing the adherence to the workshop's scope (maintenance and evolution) as well as the expectations of the categories.

We have received **22** submissions (5 CORE, 1 REAP, 6 DIVE, 10 SHOW), out of which 20 were accepted (1 CORE and 1 DIVE rejected). They came from 53 authors from 7 countries (The Netherlands, Belgium, Spain, Japan, US, Switzerland, Germany), and were reviewed by 20 programme committee members (see next page).

Vadim Zaytsev
General Chair
BENEVOL 2025

Programme Committee

- **Alexander Nolte**, Eindhoven University of Technology, Netherlands
- **Alexander Serebrenik**, Eindhoven University of Technology, Netherlands
- **Alexandre Decan**, University of Mons, Belgium
- **Ana-Maria Oprescu**, University of Amsterdam, Netherlands
- **Andrea Capiluppi**, University of Groningen, Netherlands
- **Anne Etien**, University of Lille, France
- **Carolin Brandt**, Delft University of Technology, Netherlands
- **Coen De Roover**, Vrije Universiteit Brussel, Belgium
- **Daniel Feitosa**, University of Groningen, Netherlands
- **Gregorio Robles**, Universidad Rey Juan Carlos, Spain
- **Fernando Castor**, University of Twente, Netherlands *(chair)*
- **Johan Fabry**, Raincode Labs, Belgium
- **Lina Ochoa-Venegas**, Eindhoven University of Technology, The Netherlands
- **Mairieli Wessel**, Radboud University, Netherlands
- **Marcus Gerhold**, University of Twente, Netherlands
- **Ruben Opdebeeck**, Vrije Universiteit Brussel, Belgium
- **Serge Demeyer**, Universiteit Antwerpen (ANSYMO), Belgium
- **Siamak Farshidi**, Wageningen University and Research, Netherlands
- **Slinger Jansen**, Utrecht University, Netherlands
- **Vincenzo Stoico**, Vrije Universiteit Amsterdam

Table of Contents

Section "Evolving Software Ecosystems"

Section "Software Testing"

An Introduction to Indirect Code Completion

Nhat[1], Vadim Zaytsev[1]

[1]*Formal Methods & Tools (FMT), University of Twente, Enschede, The Netherlands*

Abstract

Code completion plays a vital role in enhancing software development productivity and quality. It has evolved from simple spell checkers to advanced AI-powered tools, yet the core principle remains the same: to provide code suggestions directly where the completion was initiated. In this paper, we proposed to generalise it to *indirect code completion* (ICC), which can be initiated in one place while having suggestions proposed elsewhere. By analysing the structure and properties of ICC in the context of object-oriented languages, we identified several properties that can be used to characterise ICC. We also introduced a set of questions and a taxonomy to categorise ICC into over 20 different application patterns.

Keywords

code completion, coding context

1. Introduction and Background

The "type less, write more" idea has become familiar to most developers, with applications ranging from query completion in modern search engines based on popular queries, to character suggestions when typing in languages with thousands of characters. For code, completion, when triggered, commonly appears in the form of a list of suggestions based on the user's prompt and the surrounding context. The earliest code completion (CC) systems are merely spell checkers [1] by today's standards. Modern CC systems can also collect and prioritise suggestions of words that already appeared in the file/codebase [2]. This led to inspection systems that perform syntactic and semantic analysis on the codebase to gain an understanding of the code to provide richer and more contextually relevant completions.

Code completion has become a norm, an expectation, and a reliance for modern IDEs and code editors to have a CC system that can offer correct types, variables, fields, methods, classes, and other language-specific constructs. Today, this kind of modern code completion is commonly known as intelligent code completion, code suggestion, code prediction, auto-completion, etc., depending on the provider. It significantly enhances the coding experience and productivity, while improving code quality by increasing efficiency and accuracy, promoting discoverability, reducing cognitive load and enhancing focus, and encouraging consistency and best practices.

Machine learning (ML) and artificial intelligence (AI) have been recently and rapidly integrated into CC solutions, such as GitHub Copilot or JetBrains AI Assistant. Even for these intelligent systems, the core principle of most, if not all, CC remains the same: to provide suggestions for relevant code elements directly where the CC was initiated. For example, the following code could have three consecutive CC events, with the user-written completion prompts in blue and IDE-proposed completion edit in red:

```
1  pizzaBuilder.withCrust().withSalami().withGarlic() // RawPizza
2              .bake()|                               // BakedPizza
3
4  pizzaBuilder.withCrust().withSalami().withGarlic() // RawPizza
5              .bake().slice()|                       // ReadyPizza
6
7  pizzaBuilder.withCrust().withSalami().withGarlic()
8              .bake().slice().serve()|
```

BENEVOL'25: Belgium-Netherlands Software Evolution Workshop 2025, November 17–18, 2025, Enschede, The Netherlands
research@nhat.run (Nhat); vadim@grammarware.net (V. Zaytsev)
https://grammarware.net/ (V. Zaytsev)
0009-0004-3110-9946 (Nhat); 0000-0001-7764-4224 (V. Zaytsev)

Instead, we could think of the following one event that can produce the same result:

```
pizzaBuilder.withCrust().withSalami().withGarlic()
            .bake().slice().serve()
```

Here, in green, we denoted the target of the completion: that is, the place where the edit is aiming at. While not necessarily winning in keystrokes, this *bridge* ICC event supports the developers in letting them focus on the essentials while it can add the necessary. Another example could be an *insert* ICC event that allows the user to add another topping without moving the caret:

```
pizzaBuilder.withCrust().withSalami().withGarlic()
            .bake().slice().serve().withHam

pizzaBuilder.withCrust().withSalami().withGarlic().withHam()
            .bake().slice().serve()
```

Collectively, we call these kinds of code completion ***indirect code completion*** (ICC), as opposed to the traditional *direct code completion* (DCC) where the edit is applied directly at the location of the caret. In the following sections, we will explore in more detail how ICC can minimise developer's distractions by performing completions elsewhere without navigating to the target place first. This example merely serves as a motivation, but style-wise besides the (already popular) *builder* pattern, it is also applicable to the *fluent interface* design pattern for its extensive use of *method chaining* and *method cascading*.

While it is simple to jump to the beginning or the end of a line (e.g., using the Home or End key), moving the caret to some specific location requires intention and finesse, potentially disrupting the flow [3]. Caret movement is a frequent and basic action yet often taken for granted. It is usually done with a mouse pointer for IDEs with a graphical interface or by a combination of modes, commands, shortcuts, and controls for purely textual ones. For the latter, it is worth noting that these cursor controls in *Vim* or *Emacs* mostly involve jumping back or forth a number of characters, words, sentences, lines, or paragraphs, — all constructs of natural languages, not programming ones.

Indirect code competition provides a quick and simple way to make indirect edits in conceptually related places and reduce caret jumps while writing code. With ICC, a sequence of direct code completion and navigation events can be replaced with a single ICC event, enhancing productivity while reducing time, errors, and cognitive load. By allowing suggestions that are not bound to the initiated location, ICC can have more opportunities to complete and support the coding process.

In this paper, we present our approach and analysis of the structure and properties of indirect code completion in § 2, specifically focusing on the context of object-oriented languages. Using these properties as a base, a comprehensive taxonomy for different categories of ICC is defined. § 4 concludes the paper with some remarks and future research possibilities. Although some early evaluation of ICC has been conducted with promising results (and can be shared during the workshop), a more comprehensive evaluation is left for future work and is not included in this paper.

2. Categories of Indirect Code Completion

This section explains our approach to investigating the concept of indirect code completion for object-oriented languages. Let us define a member f of the type (class, module, package) T as $T.f(P) \to R$, if its arguments are P and its return type is R. In some cases when this is important, we will write out all arguments individually: $T.f(P_0, P_1, \dots, P_k) \to R$ for some $k \in \mathbb{N}$.

If these elements are to be chained, then we can have a series of $T_i.f_i$ for which $R_i \equiv T_{i+1}$, and write the chain down as $T_0.f_0(P_0) \to T_1.f_1(P_1) \to T_2 \dots T_n.f_n(P_n) \to R_n$ or even as $T_0.f_0 \to T_1.f_1 \to T_2 \dots T_n.f_n \to R_n$ if parameters are not important. To identify and categorise the possible kinds of ICC, let us examine the following element chain with a hypothetical ICC:

$$f_0.f_1 \dots f_i.g.f_{i+1} \dots f_n.x \dots f_{n+1} \dots f_{n'} \tag{1}$$

In the above chain (1), a completion prompt x is provided by the user in the middle of the chain after the element f_n. For simplicity's sake, the remaining elements from f_{n+1} onward are temporarily ignored. Here, x is prompted with the intention to make a completion edit g after the target element f_i.

Consider a completion edit g. In direct code completion, the completion edit is almost always closely related to the completion prompt x, typically based on the element's name. For instance, given a completion prompt `toStr`, a logical completion edit that the user aimed for could be `toString`, which starts with the same characters. However, g does not always need to be closely related to x and can be *fresh* as we have seen from the motivational example before.

The idea of ICC is based on the ability to complete a piece of code that is not directly at the position where the CC was initiated. In that sense, to be indirect, the completion edit g needs to be at a certain *distance* to the completion prompt x.

In the specific case in (1), the completion edit g is placed between the elements f_i and f_{i+1}. I.e., the *completion action* was to *insert* g after the target element f_i, leaving all original elements intact. An alternative completion action could have been to *replace* f_i with g instead of inserting it. Finally, the target element could also be used as an argument for g: this *wrapping* action results in $g(f_i)$.

Until now, an ICC only targets one element f_i. The *range* of the target, however, is not limited to one. Instead of f_i, a sequence of elements $f_i \dots f_{i+j}$ can be targeted as once, e.g., with the wrapping action:

$$f_0.f_1 \dots f_{i-1}.g(f_i \dots f_{i+j}).f_{i+j+1} \dots f_n.x| \; f_{n+1} \dots f_{n'} \tag{2}$$

Similarly, this generalisation also holds for the completion edits. A completion edit can also consist of a sequence of elements $g_0 \dots g_m$ of a *length* greater than one. An example of this with the replace action could be:

$$f_0.f_1 \dots \cancel{f_i \dots f_{i+j}} g_0 \dots g_m.f_{i+j+1} \dots f_n.x| \; f_{n+1} \dots f_{n'} \tag{3}$$

As can be observed from the above chains, the ICCs targeted elements *backward* to the completion prompt x, given the writing direction is from left to right. Let us turn focus to the previously ignored tail elements of the chain, i.e., in the *forward* direction. Similarly, all the previously defined properties also hold for the forward direction, with one exception being that the completion prompt is now effectively prepended to the tail chain. For example, the following chain is a forward version of (3):

$$f_0 \dots f_n \; x|.f_{n+1}.f_{n+2} \dots \cancel{f_i \dots f_{i+j}} g_0 \dots g_m.f_{i+j+1} \dots f_{n'} \tag{4}$$

More compactly, both backward (3) and forward (4) ICC can be represented together using the under-arrows (e.g., $\underrightarrow{x}$ for forward and $\underleftarrow{x}$ for backward) to denote the completion direction relative to where the completion was initiated, using both if both are possible starting points:

$$\dots \underrightarrow{x}|f_0.f_1 \dots \cancel{f_i \dots f_{i+j}} g_0 \dots g_m.f_{i+j+1} \dots f_n \underleftarrow{x}| \dots \tag{5}$$

Collectively, these properties (target distance, target range, completion action, completion edit freshness, completion edit length, completion direction, and depth) form the basis of the proposed ICC categorisation. To further streamline the analysis, we gathered and organised completion properties into a set of questions:

- What is the *distance* from x to the first target f_i?
- What is the *range* of the target elements?
- What is the *completion action*?
- What is the *freshness* of the completion edit?
- What is the *length* of the completion edit?
- What is the completion *direction* relative to x?

Answering these questions results in a combination of properties, which are used to categorise the ICC. These properties can also help distinguish between direct and indirect code completion. To illustrate this, let us consider a simple direct code completion $f_0.f_1 \dots f_n.x|$. Direct code completion is a special case of CC in the forward direction with a target range of only one element f_n directly before the completion prompt x. The insert action is used by direct code completion to append a completion edit x of length 1 that is closely related to the completion prompt x.

Table 1
Summary of indirect code completion categories for the pattern $\ldots \underset{\rightarrow}{x|}.f_0.f_1 \ldots f_i \ldots f_{i+j} \ldots f_n.\underset{\leftarrow}{x|} \ldots$

Compl. action	Special case	Compl. target		Compl. edit			Applied pattern
		Element	Range	Element	Param.	Length	
insert	—	f_i	1	x	—	1	$\ldots \underset{\rightarrow}{\lvert} f_0 \ldots f_i.x.f_{i+1} \ldots f_n \underset{\leftarrow}{\rvert} \ldots$
	append	f_n	1	x	—	1	$\ldots f_0 \ldots f_n.\underset{\leftrightarrow}{x\rvert} \ldots$
fill	—	f_i	1	g	—	1	$\ldots \underset{\rightarrow}{x|}.f_0 \ldots f_i.g.f_{i+1} \ldots f_n.\underset{\leftarrow}{x|} \ldots$
		f_i	1	$g_0 \ldots g_m$	—	$m+1$	$\ldots \underset{\rightarrow}{x|}.f_0 \ldots f_i.g_0 \ldots g_m.f_{i+1} \ldots f_n.\underset{\leftarrow}{x|} \ldots$
	bridge	$\underset{\rightarrow}{f_0}$ or $\underset{\leftarrow}{f_n}$	1	g	—	1	$\ldots \underset{\rightarrow}{x|.g.f_0}.f_1 \ldots f_{n-1}.\underset{\leftarrow}{f_n.g.x|} \ldots$
		$\underset{\rightarrow}{f_0}$ or $\underset{\leftarrow}{f_n}$	1	$g_0 \ldots g_m$	—	$m+1$	$\ldots \underset{\rightarrow}{x|.g_0 \ldots g_m.f_0}.f_1 \ldots f_{n-1}.\underset{\leftarrow}{f_n.g_0 \ldots g_m.x|} \ldots$
displace	—	f_i	1	x	—	1	$\ldots \underset{\rightarrow}{\lvert} f_0 \ldots f_{i-1}.\cancel{f_i} x.f_{i+1} \ldots f_n \underset{\leftarrow}{\rvert} \ldots$
		$f_i(P_i)$	1	x	P_i	1	$\ldots \underset{\rightarrow}{\lvert} f_0 \ldots f_{i-1}.\cancel{f_i} x(P_i).f_{i+1} \ldots f_n \underset{\leftarrow}{\rvert} \ldots$
		$f_i \ldots f_{i+j}$	$j+1$	x	—	1	$\ldots \underset{\rightarrow}{\lvert} f_0 \ldots f_{i-1}.\cancel{f_i \ldots f_{i+j}} x.f_{i+j+1} \ldots f_n \underset{\leftarrow}{\rvert} \ldots$
	discard	$f_0 \ldots f_n$	$n+1$	x	—	1	$\ldots \underset{\rightarrow}{\lvert} x\cancel{f_0 \ldots f_n}$ or $\cancel{f_0 \ldots f_n} x \underset{\leftarrow}{\rvert} \ldots$
replace	—	f_i	1	g	—	1	$\ldots \underset{\rightarrow}{x|}.f_0 \ldots f_{i-1}.\cancel{f_i} g.f_{i+1} \ldots f_n.\underset{\leftarrow}{x|} \ldots$
		$f_i(P_i)$	1	g	P_i	1	$\ldots \underset{\rightarrow}{x|}.f_0 \ldots f_{i-1}.\cancel{f_i} g(P_i).f_{i+1} \ldots f_n.\underset{\leftarrow}{x|} \ldots$
		f_i	1	$g_0 \ldots g_m$	—	$m+1$	$\ldots \underset{\rightarrow}{x|}.f_0 \ldots f_{i-1}.\cancel{f_i} g_0 \ldots g_m.f_{i+1} \ldots f_n.\underset{\leftarrow}{x|} \ldots$
		$f_i \ldots f_{i+j}$	$j+1$	g	—	1	$\ldots \underset{\rightarrow}{x|}.f_0 \ldots f_{i-1}.\cancel{f_i \ldots f_{i+j}} g.f_{i+j+1} \ldots f_n.\underset{\leftarrow}{x|} \ldots$
		$f_i \ldots f_{i+j}$	$j+1$	$g_0 \ldots g_m$	—	$m+1$	$\ldots \underset{\rightarrow}{x|}.f_0 \ldots f_{i-1}.\cancel{f_i \ldots f_{i+j}} g_0 \ldots g_m.f_{i+j+1} \ldots f_n.\underset{\leftarrow}{x|} \ldots$
	rewrite	$f_0 \ldots f_n$	$n+1$	g	—	1	$\ldots \underset{\rightarrow}{x|}.g\cancel{f_0 \ldots f_n}$ or $\cancel{f_0 \ldots f_n} g.\underset{\leftarrow}{x|} \ldots$
		$f_0 \ldots f_n$	$n+1$	$g_0 \ldots g_m$	—	$m+1$	$\ldots \underset{\rightarrow}{x|}.g_0 \ldots g_m\cancel{f_0 \ldots f_n}$ or $\cancel{f_0 \ldots f_n} g_0 \ldots g_m.\underset{\leftarrow}{x|} \ldots$
	expand	x	1	g	—	1	$\ldots f_0 \ldots f_n.\cancel{x} g \underset{\leftrightarrow}{\rvert} \ldots$
		x	1	$g_0 \ldots g_m$	—	$m+1$	$\ldots f_0 \ldots f_n.\cancel{x} g_0 \ldots g_m \underset{\leftrightarrow}{\rvert} \ldots$
wrap	—	f_i	1	x	f_i	1	$\ldots \underset{\rightarrow}{\lvert} f_0 \ldots f_{i-1}.x(f_i).f_{i+1} \ldots f_n \underset{\leftarrow}{\rvert} \ldots$
		$f_i \ldots f_{i+j}$	$j+1$	x	$f_i \ldots f_{i+j}$	1	$\ldots \underset{\rightarrow}{\lvert} f_0 \ldots f_{i-1}.x(f_i \ldots f_{i+j}).f_{i+j+1} \ldots f_n \underset{\leftarrow}{\rvert} \ldots$
	surround	$f_0 \ldots f_n$	$n+1$	x	$f_0 \ldots f_n$	1	$\ldots \underset{\rightarrow}{\lvert} x(f_0 \ldots f_n)$ or $x(f_0 \ldots f_n) \underset{\leftarrow}{\rvert} \ldots$
reshape	—	f_i	1	g	f_i	1	$\ldots \underset{\rightarrow}{x|}.f_0 \ldots f_{i-1}.g(f_i).f_{i+1} \ldots f_n.\underset{\leftarrow}{x|} \ldots$
		$f_i \ldots f_{i+j}$	$j+1$	g	$f_i \ldots f_{i+j}$	1	$\ldots \underset{\rightarrow}{x|}.f_0 \ldots f_{i-1}.g(f_i \ldots f_{i+j}).f_{i+j+1} \ldots f_n.\underset{\leftarrow}{x|} \ldots$
	adapt	$f_0 \ldots f_n$	$n+1$	g	$f_0 \ldots f_n$	1	$\ldots \underset{\rightarrow}{x|}.g(f_0 \ldots f_n)$ or $g(f_0 \ldots f_n).\underset{\leftarrow}{x|} \ldots$
	fit	x	1	g	x	1	$\ldots f_0 \ldots f_n.\underset{\leftrightarrow}{g(x)\rvert} \ldots$

Tab. 1 summarises the different categories of ICC: we primarily categorise ICCs by the completion action, then the targeted element(s), and by the completion edit. In some categories, special cases can be identified by a certain combination of properties. For each category, there can be some variants that differ in the distance, range, length, and direction of the completion.

Let us take a closer look at *Insert*, one of the simplest kinds of ICC. In this category, a completion edit x closely related to the completion prompt x is inserted into the chain after the target element f_i, leaving all the original elements in the chain intact. An example of *insert* ICC was shown in the introduction. More formally, an *insert* ICC can be written as $f_0.f_1 \ldots f_i.x.f_{i+1} \ldots f_n.|f_{n+1} \ldots f_{n'}$.

Here, x can only be chained between f_i and f_{i+1}, if the insertion of x can maintain the type flow of the chain, i.e., x must also be a member of R_i and f_{i+1} is a member of T_x. When $i = n$, the completion edit x is inserted at the exact place where the completion was initiated, effectively a direct code completion.

One can analyse and define other categories of ICC similarly, which, in the interest of space, we will leave up to the reader. As for a brief overview, *fill* is similar to *insert*, but the completion edits are *fresh* elements $g/g_0 \dots g_m$. As a special case of *fill*, *bridge* fills the gap between the completion prompt x and the adjacent target element, as was also seen in the motivating example. Instead of just inserting an element after, *displace* and *replace* also remove the target element(s), effectively displacing/replacing them with the completion edit x or $g/g_0 \dots g_m$ respectively, optionally inherit the parameter P_i of the target. *Discard* and *rewrite* are special cases of *displace* and *replace* that remove the entire head or tail, depending on the completion direction. *Expand* is another special case of *replace* that targets the prompt x itself. A different pair of completion actions are *wrap* and *reshape*; Instead of removing, they consider the target element(s) as arguments for their completion edit x or g respectively, with the special cases *surround* and *adapt* that targeting the entire head or tail, depending on the completion direction. An even more special case *fit* that targets the completion prompt x itself.

3. Related Work

Numerous approaches have been proposed to expand code completion in various dimensions [4, 5, 6]. However, none of these studies investigate the idea of completing the code indirectly. The closest approaches to ICC are template-based completion techniques.

One of the most common template-based completion techniques is code snippet generation/completion. In *Visual Studio (Code)*, snippets [7] are part of *IntelliSense* [8]. In *JetBrains' IDEs*, it is called *Live templates* [9]. Regardless of the branding, the idea of snippet generation stays the same: replace an abbreviated predefined "template identifier" with a predefined template. E.g., with the prompt `fori`, the completion system will replace this with a predefined template `for (int i = 0; i < ?; i++) {...}`. Snippet templates are typically repeating code patterns that can be defined to reduce coding effort.

In a similar manner, *JetBrains' postfix completion* [10] allows predefined templates to be used on an expression via an abbreviated postfix. For example, given an expression `isActive()` that can be evaluated to an expression `boolean`, the postfix completion `isActive().if` matches the predefined template `if($EXPR$) {...}` with template identifier `if` postfixed to an `boolean` expression, resulting in the completion `if(isActive()) {...}`.

Conceptually, these template-based completion techniques resemble *expand* ICC, replacing the completion prompt with a completion edit. If the template is closely related to the completion prompt, it can also be considered as a case of *append*. Nevertheless, these techniques only use predefined templates, individually established, and do not actively offer code completion suggestions as ICC does.

4. Conclusion and Future Work

In this paper, we have introduced *indirect code completion (ICC)* — a novel approach that allows code completion to make edits to other existing elements that are not restricted to the location where the completion was initiated. Analysis on the structure and properties of ICC in the context of object-oriented languages led to the categorisation of different application patterns. Each category can further be formalised to define the requirements and constraints that could serve as the foundation and aid the design and implementation of ICC.

ICC opens the door to more powerful and complex suggestions that could greatly increase productivity and code quality. New research opportunities could explore what could be ICC suggestion candidates, better-fitting searching algorithms, or ICC in other paradigms. Advancements in the field of (direct) code completion and AI can also be applied to ICC to provide richer and more relevant suggestions. It also remains to be seen which theoretically possible ICC kinds are practically useful. Early evaluation using the *MSR 2018* [11] dataset returned promising results [6], where a considerable number of equivalent sequences of direct code completion and other interaction events can be replaced by a single ICC. Our current focus is on developing an implementation of ICC so that more accurate evaluation can be done in future studies.

Declaration on Generative AI

The authors have not employed any Generative AI tools to create, change or rephrase the content of this document.

References

[1] J. L. Peterson, Computer Programs for Detecting and Correcting Spelling Errors, Communications of the ACM 23 (1980) 676–687. doi:10.1145/359038.359041.

[2] M. Asaduzzaman, C. K. Roy, K. A. Schneider, D. Hou, CSCC: Simple, Efficient, Context Sensitive Code Completion, in: Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2014, pp. 71–80. doi:10.1109/ICSME.2014.29.

[3] S. Janssens, V. Zaytsev, Go with the Flow: Software Engineers and Distractions, in: T. Kühn, V. Sousa, S. Abrahão, T. C. Lethbridge, E. Renaux, B. Selić (Eds.), MoDELS'22 Companion Proceedings: Sixth International Workshop on Human Factors in Modeling / Modeling of Human Factors (HuFaMo), 2022, pp. 934–938. doi:10.1145/3550356.3559101.

[4] L. L. Nunes da Silva Jr., T. Nazareth de Oliveira, A. Plastino, L. G. P. Murta, Vertical Code Completion: Going Beyond the Current Ctrl+Space, in: Proceedings of the Sixth Brazilian Symposium on Software Components (SBCARS): Architectures and Reuse, IEEE CS, 2012, pp. 81–90. doi:10.1109/SBCARS.2012.22.

[5] Y. Y. Lee, S. Harwell, S. Khurshid, D. Marinov, Temporal Code Completion and Navigation, in: Proceedings of the 35th International Conference on Software Engineering (ICSE), IEEE CS, 2013, pp. 1181–1184. doi:10.1109/ICSE.2013.6606673.

[6] Nhat, V. Zaytsev, CoCoCoLa: Code Completion Control Language, in: Proceedings of the 24th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE), ACM, 2025, p. 1–12. doi:10.1145/3742876.3742883.

[7] Microsoft, Snippets in Visual Studio Code, 2025. URL: https://code.visualstudio.com/docs/editing/userdefinedsnippets.

[8] Microsoft, IntelliSense in Visual Studio, 2025. URL: https://learn.microsoft.com/en-us/visualstudio/ide/using-intellisense.

[9] IntelliJ, Live templates | IntelliJ IDEA Documentaion, 2025. URL: https://www.jetbrains.com/help/idea/using-live-templates.html.

[10] IntelliJ, Postfix code completion | IntelliJ IDEA Documentaion, 2025. URL: https://www.jetbrains.com/help/idea/postfix-code-completion.html.

[11] International Conference on Mining Software Repositories (MSR), MSR 2018 — Mining Challenge, 2018. URL: https://2018.msrconf.org/track/msr-2018-Mining-Challenge.

Same Size, Different Costs: Phase-Level Energy Variations in Transformer Models during Code Generation

Lola Solovyeva[1]

[1]University of Twente, Enschede, the Netherlands

Abstract

AI-assisted tools are increasingly integrated into software development, augmenting workflows in code generation, bug fixing, testing, and documentation. However, their inference introduces extra energy costs that affect the sustainability of the software lifecycle. In this study, we measure phase-level energy consumption of LLMs, focusing on four transformer models of comparable size using HumanEval dataset for code generation under different batch sizes. Our findings show that models with similar parameter counts exhibit distinct energy consumption patterns across prefill and decoding phases. These results highlight that LLMs of the same architecture type and with similar parameter counts can still differ due to low-level implementation details, which should be considered when developing strategies to reduce energy consumption in software development.

1. Introduction

AI-assisted tools are increasingly integrated into software development processes [1]. In the context of software maintenance and evolution, these tools augment developer workflows in scenarios such as code generation, refactoring, bug detection, and testing [2]. While LLMs can accelerate those tasks, their inference processes introduce a non-trivial energy cost, particularly when used repeatedly in CI/CD pipelines or large-scale maintenance workflows. Research [3] suggested that OpenAI required 3,617 of NVIDIA's HGX A100 servers, with a tottal of 28,936 GPUs , to support ChatGPT, implying that it requires 564 MWh per day for its inference. Meanwhile, an estimate of 1,287 MWh was used in GPT-3 training phase. As a result, the overall sustainability of the software lifecycle now also depends on the efficiency of the AI tools that support them. While existing studies on LLM efficiency focus on architectural techniques, these approaches often treat inference as a uniform process [4]. In practice, inference consists of two distinct phases: **prefill**, that processes the input prompt and generates internal key/value representations (compute-bound), and **decoding**, that generates output tokens autoregressively using these cached representations (memory-bound).

In this work, we demonstrate that transformer models of similar sizes exhibit distinct energy consumption patterns across both phases. Hence, reducing the overall energy consumption of their inference requires model-specific optimization strategies.

2. Methodology

To record energy measurements per phase, we adopted the method originally proposed by Babakol et al. [5]. The method involves two parallel processes: (1) collecting GPU energy samples with pyNVML every 0.01 seconds along with their timestamps, and (2) recording timestamps at the start and end of generating each token. There was no other process running on the same GPU. The timestamps are then aligned to measure the energy consumption of each phase.

Four widely used transformer models with roughly similar parameter counts were selected from Hugging Face, ensuring they could be accommodated on an NVIDIA A10 GPU (24 GB RAM): Llama3.2 (3B), Qwen2.5-Coder (3B), Gemma3 (4B), and Phi3.5 (4B). We used the HumanEval dataset for code

BENEVOL'25: Proceedings of the 24th Belgium-Netherlands Software Evolution Workshop, 17–18 November 2025, Enschede, The Netherlands

o.solovyeva@utwente.nl (L. Solovyeva)

0009-0008-6903-7086 (L. Solovyeva)

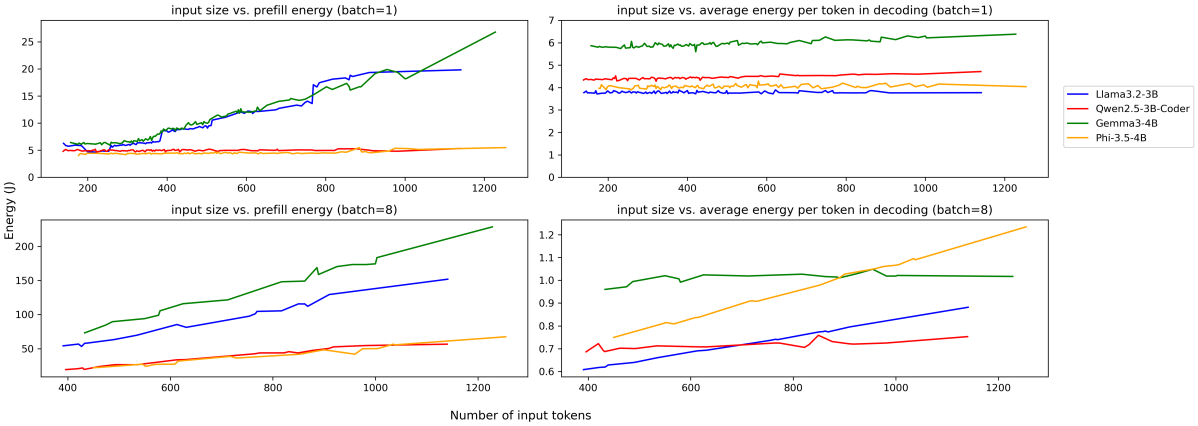


Figure 1: Influence of input tokens and batching on prefill phase costs and per-token costs in the decoding phase.

generation and evaluated the models with batch sizes of 1 and 8 to examine how a single request and batching, that increases the workload of the model, influence both phases.

3. Findings & Implications

Figure 1 shows the relationship between input prompt size and its impact on the prefill phase as well as per-token energy during the decoding stage for the models in this study. Overall, larger prompts and increased batch sizes lead to higher prefill costs. However, the magnitude of this increase varies across models, with some showing greater sensitivity to input size. Llama3.2 (3B) and Gemma3 (4B) exhibit a steeper increase compared to Qwen2.5 (3B) and Phi3.5 (4B), despite having similar parameter counts.

In regard to the influence of input size on the decoding stage, we can observe expected differences in costs between the models for a single request, since larger models would exhibit higher costs. A more interesting pattern emerges when processing batched requests, which increases the workload by combining multiple requests in one. The models respond differently: Phi3.5 (4B) and Llama3.2 (3B) show approximately a 1.5×increase in energy per token when the input grows from 400 to 1200 tokens, whereas the other two models are either unaffected or exhibit a much smaller increase.

These findings suggest that even among models of the same architecture type with similar parameter counts, their energy patterns differ across phases, indicating that these differences likely stem from low-level implementation details such as memory management and runtime optimizations. Furthermore, the choice of model within the software development lifecycle should depend on the specific task. For example, models that are less sensitive to input size may be better suited for tasks involving larger inputs, such as code translation, test or docstring generations.

References

[1] C. Ebert, P. Louridas, Generative ai for software practitioners, IEEE Software 40 (2023) 30–38. doi:10.1109/MS.2023.3265877.

[2] N. Alizadeh, B. Belchev, N. Saurabh, P. Kelbert, F. Castor, Language models in software development tasks: An experimental analysis of energy and accuracy, 2025. URL: https://arxiv.org/abs/2412.00329.

[3] A. de Vries, The growing energy footprint of artificial intelligence, Joule 7 (2023) 2191–2194. doi:https://doi.org/10.1016/j.joule.2023.09.004.

[4] M. F. Argerich, M. Patiño-Martínez, Measuring and improving the energy efficiency of large language models inference, IEEE Access 12 (2024) 80194–80207. doi:10.1109/ACCESS.2024.3409745.

[5] T. Babakol, Y. D. Liu, Tensor-aware energy accounting, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24, Association for Computing Machinery, New York, NY, USA, 2024. URL: https://doi.org/10.1145/3597503.3639156.

The Cost of AI-Assisted Coding: Energy vs. Accuracy in Language Models*

Negar Alizadeh[1], Boris Belchev[2], Nishant Saurabh[1] and Patricia Kelbert[3]

[1]Utrecht University, Utrecht, The Netherlands
[2]University of Twente, Enschede, The Netherlands
[3]Fraunhofer IESE, Kaiserslautern, Germany

1. Introduction and Motivation

Generative Large Language Models (LLMs) have become widely accessible since the release of ChatGPT in late 2022 [2], and their adoption nearly doubled in under six months [3]. In addition, the majority of developers find code-specific AI models beneficial and have integrated them into their daily workflows. [4, 5].

Even though these AI tools are accessible through third-party APIs, client companies are mainly concerned about data privacy, security, and subscription costs. This motivates the use of locally deployed open-access language models. One approach for deploying LLMs locally is to utilize a flagship GPU with sufficient memory optimized for Deep Learning (DL) applications and to set up an open-access LLM on it. However, since these GPUs are not affordable for everyone, an alternative solution could be using compressed and quantized models that can run on smaller GPUs or even large CPUs.

At the same time, the energy cost of LLMs, especially during inference, has become a growing concern due to financial and environmental impacts [6, 7]. Furthermore, recent studies evaluating code-centric LLMs have mainly focused on performance in terms of accuracy, often overlooking their energy footprint.[8, 9].

The goal of this study is to investigate the energy consumption of using LLMs during the inference phase in typical software development tasks, namely code generation, bug fixing, docstring generation, and test case generation. We selected these 4 tasks because they are widely used by developers [10] and frequently addressed in software engineering research involving deep learning [11]. They also represent the software development lifecycle, from implementation and documentation to testing and maintenance.

In particular, we address the following research questions:

RQ1 How does energy usage vary across software-related tasks?

RQ2 Is there a trade-off between energy efficiency and accuracy?

RQ3 Which model characteristics influence energy consumption?

RQ4 What are the performance differences between general-purpose models and code-specific models?

2. Methodology and Results

Our experimental design involves evaluating 18 language model families across three precision formats on two real-world hardware setups: a high-end A100 GPU and a consumer-grade RTX3070 GPU. Model

BENEVOL'25: Proceedings of the 24th Belgium-Netherlands Software Evolution Workshop, 17–18 November 2025, Enschede, The Netherlands
*Accepted at the 22nd International Conference on Mining Software Repositories (MSR 2025) [1]
n.s.alizadeh@uu.nl (N. Alizadeh); b.belchev@student.utwente.nl (B. Belchev); n.saurabh@uu.nl (N. Saurabh); patricia.kelbert@iese.fraunhofer.de (P. Kelbert)

selection was based on their popularity on Hugging Face, availability, and the reputability of their creators. Running an LLM locally requires a compatible runtime to handle inference across devices. We chose Ollama based on its popularity on GitHub and ease of switching between models. All evaluations are conducted on the HumanEvalPack dataset [12], with top-p set to 0.95 and temperature to 0.1, following prior studies [13, 14, 15]. As for the evaluation metric, we report test coverage and correctness for test generation and pass@1 for the rest.

Our findings indicate that a model's energy consumption is directly affected by the software development task it performs, with notable performance variations between tasks. Therefore, selecting models based on the tasks they are expected to handle is key to reducing the energy footprint. In contrast, energy usage per generated token remains consistent across tasks for a model. Additionally, the total energy consumed by each model is strongly correlated with its architectural characteristics, suggesting that some aspects of an LLM's architecture can help estimate its efficiency, given that the average size of the outputs can be anticipated. Finally, we observed that energy consumption and accuracy do not always require a compromise, as larger models often have a significantly higher energy footprint while performing similarly, or even being outperformed by smaller models in terms of accuracy. Finally, "Coding"-specific models can be more accurately described as "code generation" models. Our results suggest that fine-tuning models for other tasks, such as docstring generation and bug fixing, is a potential research avenue.

To improve generalizability, future work could include more programming languages and use benchmarks with complex, real-world tasks rather than simple Python functions. Currently, we are exploring ways to predict model efficiency based on architectural features, to reduce the need for costly experiments.

References

[1] N. Alizadeh, B. Belchev, N. Saurabh, P. Kelbert, F. Castor, Language models in software development tasks: An experimental analysis of energy and accuracy, in: 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR), IEEE, 2025, pp. 725–736. doi:10.1109/MSR66628.2025.00109.

[2] C. Ebert, P. Louridas, Generative AI for software practitioners, IEEE Softw. 40 (2023) 30–38. doi:10.1109/MS.2023.3265877.

[3] Microsoft Corporation, AI at Work Is Here—Now Comes the Hard Part, "https://www.microsoft.com/en-us/worklab/work-trend-index/ai-at-work-is-here-now-comes-the-hard-part", 2024. Accessed: 2024-11-05.

[4] Stack Overflow, Developers get by with a little help from ai: Stack overflow knows code - assistant pulse survey results, https://stackoverflow.blog/2024/05/29/developers-get-by-with-a-little-help-from-ai-stack-overflow-knows-code-assistant-pulse-survey-results/, 2024. Accessed: 2024-11-08.

[5] GitHub, Research: Quantifying github copilot's impact in the enterprise with accenture, https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-in-the-enterprise-with-accenture/, 2024. Accessed: 2024-11-08.

[6] E. Strubell, A. Ganesh, A. McCallum, Energy and policy considerations for deep learning in NLP, CoRR abs/1906.02243 (2019).

[7] A. Lacoste, A. Luccioni, V. Schmidt, T. Dandres, Quantifying the carbon emissions of machine learning, CoRR abs/1910.09700 (2019).

[8] Z. Zheng, K. Ning, Y. Wang, J. Zhang, D. Zheng, M. Ye, J. Chen, A survey of large language models for code: Evolution, benchmarking, and future trends, CoRR abs/2311.10372 (2023).

[9] J. Jiang, F. Wang, J. Shen, S. Kim, S. Kim, A survey on large language models for code generation, CoRR abs/2406.00515 (2024).

[10] M. Khemka, B. Houck, Toward effective AI support for developers: A survey of desires and concerns, Commun. ACM 67 (2024) 42–49. doi:10.1145/3690928.

[11] C. Watson, N. Cooper, D. Nader-Palacio, K. Moran, D. Poshyvanyk, A systematic literature review

on the use of deep learning in software engineering research, ACM Trans. Softw. Eng. Methodol. 31 (2022) 32:1–32:58. doi:10.1145/3485275.

[12] N. Muennighoff, Q. Liu, A. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. von Werra, S. Longpre, Octopack: Instruction tuning code large language models, CoRR abs/2308.07124 (2023).

[13] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, et al., Starcoder: may the source be with you!, Trans. Mach. Learn. Res. 2023 (2023).

[14] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, et al., Starcoder 2 and the stack v2: The next generation, CoRR abs/2402.19173 (2024).

[15] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. Canton-Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, G. Synnaeve, Code llama: Open foundation models for code, CoRR abs/2308.12950 (2023).

Bridging CPU and GPU in Rust

Niek Aukes[1], Cristian-Andrei Begu[1] and Georgiana Caltais[1]

[1]University of Twente, The Netherlands

Abstract

As heterogeneous computing becomes widespread, fragmented workflows for managing separate CPU and GPU codebases create significant challenges for software evolution. This paper introduces a unified compilation model for Rust that addresses this fragmentation by treating GPU kernels as native functions. Using a prototype compiler, we demonstrate the benefits of treating GPU kernels as native code with a case study: migrating an existing multi-threaded CPU program to use GPU acceleration. We introduce a unified compilation model that contributes to reducing maintenance overhead and technical debt in migrating and evolving heterogeneous systems.

Keywords

Rust, compiler, hybrid compilation, GPU acceleration, CUDA, heterogeneous programming

1. Introduction

Heterogeneous computing, in which a CPU orchestrates GPUs executing highly parallel kernels, has become widespread across various disciplines, including scientific simulation, real-time rendering, and emerging artificial intelligence workloads. This is reflected in real-world adoption: the June 2025 TOP500 list [1] indicates that over one-third of high-performance systems use GPU accelerators. Yet, despite this hardware revolution, the software infrastructure remains fragmented: host code is traditionally written in C, C++ or Python, while performance-critical kernels are implemented in specialized languages like CUDA C [2] or OpenCL C [3]. This divide may result in developers maintaining parallel code bases, which may lead to increased maintenance costs. GPU-specific languages further suffer from a narrower ecosystem: they lack the rich standard libraries and package-management facilities of mainstream CPU languages. Consequently, even light refactoring requires edits in two languages, validation across two toolchains, and custom build steps. These fractured workflows may not only hinder development but also incur technical debt, undermining the maintainability and long-term evolution of heterogeneous applications.

Several tools have attempted to reduce this complexity. In C and C++, OpenMP [4] provides directives that allow loops to be parallelized on GPU hardware via annotations. This lowers the entry cost compared to writing explicit CUDA or OpenCL, but the model is applicable to loop-based parallel patterns and offers less control over execution and optimization than explicit GPU programming. High-level languages have also partially addressed the divide: Numba [5], for example, compiles annotated Python functions into CUDA kernels, offering in-place acceleration within a single Python syntax and integration with the broader Python stack. While performance limitations from Python's dynamic typing, interpreted nature, and the global interpreter lock primarily affect non-accelerated code, Numba introduces its own challenges. Its programming model is often non-idiomatic for Python developers, and its error messages can be difficult to understand. Compiled languages have fared no better: GPU programs typically resort to OpenCL APIs, while NVIDIA's CUDA C++ compiler still requires a two-stage build that extracts and compiles kernels separately [6]. These approaches leave developers with split toolchains and additional build complexity.

Despite these challenges, Rust's combination of zero-cost abstractions, a compile-time ownership model and borrow checker (the compiler enforcing safe memory sharing rules), and a comprehensive standard library [7] positions it as an ideal base for unified heterogeneous programming. These checks,

BENEVOL25: The 24th Belgium-Netherlands Software Evolution Workshop, Autumn 2025, Enschede, The Netherlands
n.aukes@student.utwente.nl (N. Aukes); c.begu@student.utwente.nl (C. Begu); g.g.c.caltais@utwente.nl (G. Caltais)


together with a strong static type system, catch many concurrency and memory safety errors at compile time, while Cargo and `crates.io` provide a mature package ecosystem with over 190,000 packages [8]. Prior work by Holk et al. [9] already demonstrated the possibility of extending Rust with GPU support. Their prototype showed that Rust could be used as a base for hybrid compilation, by using a pre-processing step to separate host and device code.

More recently, projects such as Rust-GPU [10] and Rust-CUDA [11] have advanced the state of GPU programming in Rust by showing that safe, idiomatic Rust can directly target GPU execution environments. Their progress highlights the potential for a convergent toolchain where unified compilation and GPU-first abstractions reinforce each other. Alongside this, Faé and Griebler developed a macro-based GPU accelerator [12] demonstrating an alternative path, trading language flexibility for a simplified experience. These efforts point toward a growing ecosystem where our compiler can interoperate and extend the reach of Rust on heterogeneous platforms.

Our contribution is to showcase a unified compilation approach for Rust that treats GPU kernels as native Rust functions, building on prior compiler efforts. This reduces the need for separate kernel languages and build steps, making it easier for developers to write, migrate, refactor, and test heterogeneous applications within a familiar Rust workflow.

The remainder of this paper is organized as follows. Section 2 presents the system design, detailing the approach that enables unified CPU–GPU compilation. Section 3 demonstrates the proposal through a case study: migrating, testing, and evolving an existing Rust program using the hybrid compiler. Section 4 discusses the implications of our compiler, its limitations, and potential directions for the future.

2. The Unified Compiler

In this section, we describe our unified compiler and the changes made to the Rust compilation pipeline to treat GPU kernels as native Rust functions, without disrupting the original host workflow. A high-level overview of the unified compiler is shown in Figure 1. For an in-depth description of its design and implementation, we refer readers to our earlier work [13, 14]. The source code, along with installation instructions, is available at https://github.com/NiekAukes/rust-gpu-hybrid-compiler

In standard Rust, the compiler parses source code into an Abstract Syntax Tree, lowers it to the High-level Intermediate Representation for type checking, then to the Mid-level Intermediate Representation (MIR) for borrow checking and optimizations, and finally emits LLVM IR for machine code generation [15]. The compiler executes these stages via "queries", on-demand computations that retrieve information such as a variable's type or the body of a function when required.

The design of Holk et al. [9] introduces a pre-processing step that identifies functions marked with the `#[kernel]` attribute and generates two separate source code artifacts: one for the host and one for the device. The host code is compiled with the standard Rust compiler, while the device code is handled

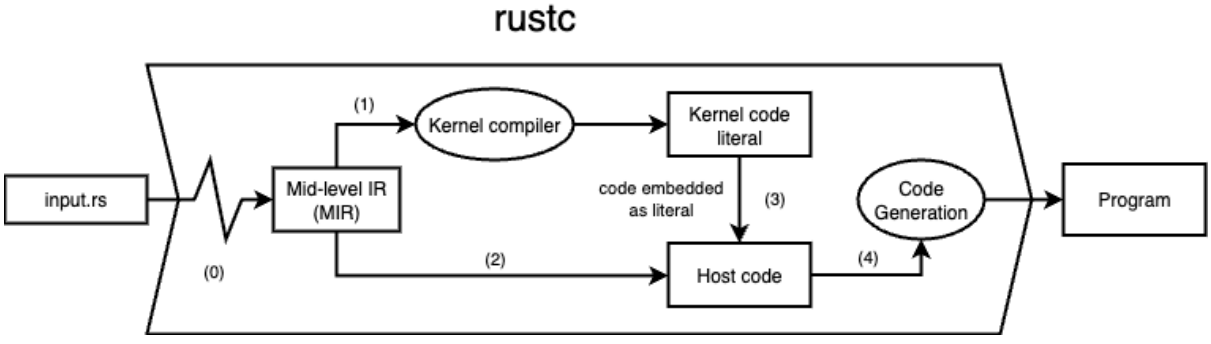


Figure 1: Compilation of annotated functions: MIR splits into GPU (1-3) and CPU (2) paths, converging in LLVM backend (4); parsing, type, and borrow checks are shared (0)

by a modified Rust compiler that emits GPU bytecode. These two outputs are then re-integrated at runtime via OpenCL bindings.

In our approach, functions marked with a `#[kernel]` attribute trigger the compiler to fork the MIR into two paths. As illustrated in Figure 1, the GPU path (1) lowers the MIR into GPU bytecode, which is then embedded back into the CPU MIR as a static object (3). The CPU path (2) continues with both the host logic and the embedded GPU code. From there, the combined MIR is passed into the standard LLVM backend for code generation (4), producing a single executable that contains both CPU and GPU components. This design allows all parsing, type checking, and borrow checking (0) to remain shared, while the final binary is produced by a single toolchain from a single source.

Our modified compiler preserves the full set of Rust's core language features[1], including type checking, borrow checking, lifetimes, generics, and traits. Beyond these, it also supports several advanced capabilities, such as defining custom panic (exception) handlers and performing dynamic memory allocation directly on the GPU. All supported features should behave consistently with their CPU counterparts, which provides several advantages: developers can reuse the same source code across CPU and GPU, and rely on identical execution semantics. Importantly, our compiler retains full compatibility with existing Rust projects and tooling.

Although our model has many benefits, there are also limitations with the current implementation of the programming model. Most importantly, only CUDA-enabled GPUs are currently supported by our compiler. This is because the architectural design of Nvidia GPUs allows many CPU execution constructs that are not available on other platforms. Examples include dynamic dispatch of functions (object-oriented programming) and function recursion. CUDAs open-specification counterpart, Vulkan [16], simply do not support these constructs that the Rust Compiler relies on [17, ch.2.16].

Beyond traditional models, it is also useful to contrast our work with ongoing efforts in the Rust ecosystem. Rust-GPU targets shader and kernel development and focuses on compiling Rust directly into GPU targets [11, 10]. While it enables a safe, expressive programming model for GPU code, it still requires complex build steps and careful separation between CPU, GPU, and shared logic [18]. By contrast, our approach emphasizes single-source development and unifies CPU and GPU code within one compiler pipeline, which simplifies maintenance and testing across heterogeneous backends.

Another approach is that of the macro-based accelerator[12], which provides a higher-level abstraction by transforming constrained Rust code into GPU-executable code through procedural macros. This model reduces developer burden, but it achieves this simplicity by enforcing restrictions on how GPU code is written. Our compiler takes the opposite stance: it supports the full Rust core language and many advanced features within GPU kernels, while preserving execution consistency between CPU and GPU paths.

3. A Mandelbrot Example

To showcase the potential of our hybrid compilation approach, we adapt an existing open-source Mandelbrot set generator, originally a traditional multi-threaded CPU program in Rust, to our modified compiler. The Mandelbrot set (Figure 2) is a well-known example in parallel computing: a fractal rendered by applying the same iterative computation independently to each pixel of the image, making it an ideal workload for both CPU parallelism and GPU acceleration.

The original project is available at https://github.com/JohnTWilkinson/Gendelbrot and our migrated version is available at https://github.com/NiekAukes/Gendelbrot. By working with an existing codebase, we provide a supporting example that illustrates how our approach can introduce GPU

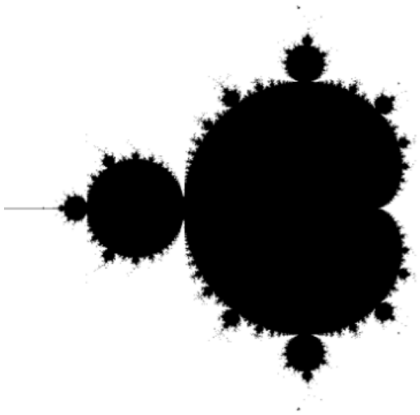


Figure 2: A Mandelbrot fractal.

[1]Core features include all language constructs and the core library, but exclude the standard library

acceleration into software not originally designed for heterogeneous execution.

The first stage in the migration moves the computation from CPU threads to GPU execution. In a conventional CUDA or OpenCL port, the main computation function would be rewritten in a GPU-specific language, moved into a separate module, compiled with a separate toolchain, and manually integrated with the host code. In contrast, our migration, shown in Figure 3, keeps the computation entirely in Rust. The existing threaded loop (lines 1–18, left) is replaced by a GPU kernel (lines 1–17, right). Instead of spawning threads explicitly (line 2, left), the kernel is launched once from the host (lines 19–21, right). The nested loops over pixel coordinates (lines 4–5, left) are replaced by a computation based on the thread index (lines 5–7, right), which directly maps each GPU thread to a pixel. The coordinate calculation (lines 6–8, left vs. lines 9–11, right) remains the same, and the `Complex` type used in both versions refers to the same type definition (line 10, left vs. line 13, right), reused directly inside the GPU kernel. The image update (lines 11–13, left vs. lines 13–15, right) also maps directly between the two versions. The rest of the codebase, including its use of Rust libraries and the Cargo build system, remain identical.

Compared with traditional migration approaches, our model sits at a middle ground between OpenMP and OpenCL. Like OpenMP, the changes required are extensions within the same source language, which avoids splitting the program into separate codebases. At the same time, similar to OpenCL, developers must still understand the parallel execution model: loops do not transform automatically but are re-expressed through thread identifiers. In addition, knowledge of GPU primitives such as synchronization and thread grouping, remains important when adapting more complex computations.

```
1  for t in 0..threads {
2      thread::spawn(move || {
3          let mut slice = vec![u8::MAX;
           ↪ this_height * image_width];
4          for i in 0..this_height {
5          for j in 0..image_width {
6              let x = real_start +
7                      (j * real_step);
8              let y = i_start - (i * i_step);
9
10             let p = Complex::new(x, y);
11             if p.is_stable(iterations) {
12                 slice[i * image_width + j] = 0;
13             }
14         }
15         }
16         tx.send((t, slice)).unwrap();
17     });
18 }
```

```
1  #[kernel]
2  fn mandelbrot(
3      mut image: Buffer<u8>, offset: usize, ...
4  ) {
5      let pos = offset + gpu::global_tid_x();
6      let i = pos / image_width;
7      let j = pos % image_width;
8
9      let x = real_start +
10             (j * real_step);
11     let y = i_start - (i * i_step);
12
13     let p = Complex::new(x, y);
14     if p.is_stable(iterations) {
15         image.set(i * image_width + j, 0);
16     }
17 }
18
19 mandelbrot.launch(
20     threads, blocks, image, 0, ...
21 ).unwrap();
```

Figure 3: Migration from CPU threads (left) to GPU kernel (right).

An additional benefit during migration is that testing requires no changes to existing workflows. Since the hybrid compiler allows GPU kernels to call the same logic as the CPU path, existing unit tests, such as those checking the behavior of the `Complex` type, run without modification within Cargo's testing framework. Beyond these unchanged unit tests, the unified workflow makes it trivial to add regression tests that compare the outputs of the CPU and GPU implementations. Such comparisons are more complex in a split-language environment, where separate toolchains make it impractical to compare host and device code in a single test suite.

After migrating Gendelbrot, we validated the new implementation using Rust's built-in testing framework [7, ch.11]. Before migration, Gendelbrot already included a suite of unit tests testing the CPU implementation of the algorithm. During migration, we preserved all of these existing tests without modification, since the hybrid compiler allows GPU kernels to invoke the same logic as the

```rust
#[test]
fn test_mandelbrot_gpu_simple_default() {
    let options = MandelbrotCpu::default();
    let image = build_mandelbrot_gpu_simple(&options);
    assert_eq!(image.len(), options.image_width * options.image_height);
    let expected_image = build_mandelbrot_cpu_simple(&options);

    if image != expected_image {
        println!("GPU image does not match expected output.");
        export_image(&image, options.image_width, options.image_height, "gpu_output.png");
        export_image(&expected_image, options.image_width, options.image_height,
        ↪  "expected_output.png");
        assert!(false, "GPU image does not match expected output.");
    }
}
```

Figure 4: Example of testing the GPU implementation against the verified CPU implementation

CPU path. In addition, we extended the suite with regression tests that directly compare the output of the GPU execution against the verified CPU baseline (Figure 4). These tests also provide a systematic way to detect potential divergences with further optimization or refactoring.

Looking forward, the single-source model also has implications for long-term evolution. In a traditional CUDA or OpenCL program, every structural or behavioral change requires coordination and mirroring between host and device definitions, often across different languages and toolchains. This duplication can add extra work during refactoring and may increase the likelihood of inconsistencies, such as mismatched data layouts, incomplete updates to kernel logic, or divergent type definitions. By ensuring that both CPU and GPU code operate on the same shared structures and are validated in the same compiler pass, our approach should reduce this type of user errors. Our approach should also reduce technical debt and enable heterogeneous programs to evolve similarly to their CPU-only counterparts.

4. Conclusion

We presented a single-source compilation model that facilitates the evolution of existing Rust programs into heterogeneous CPU/GPU applications with minimal code changes. This model operates within the standard Cargo ecosystem, enabling developers to leverage familiar tooling and a large package registry. Our approach also improves long-term maintainability by allowing host and device code to share data structures and logic, which simplifies future refactoring efforts. However, this approach introduces its own set of challenges. Currenty, our work only supports CUDA-enabled GPUs, which makes the current work impractical for cross-platform programming. Supporting cross-platform tools like Vulkan remains challenging, but possible. Additionally, the reliance on a modified compiler requires dedicated maintenance to keep pace with the official Rust toolchain's evolution.

Several paths for future work could improve the model's viability. A performance analysis against established toolchains could guide the implementation of GPU-specific optimization passes. Additional developer support could be provided through GPU-specific lints that detect anti-patterns, which could be integrated into Cargo's workflow. Moving beyond the current NVIDIA-only backend toward cross-platform APIs like Vulkan would greatly increase the compiler's relevance. Collaborating with initiatives such as Rust-GPU [10] and Rust-CUDA [11] could play a pivotal role in achieving this. Refactoring tools could also be developed to help automate the migration of existing CPU code. Finally, exploring ways to reduce the maintenance of the compiler fork, such as through a more modular architecture, would help with its maintainability.

Acknowledgements. This work is partially supported by the CYCLIC project (file no. OCENW.XL.23.089) of the research programme Open Competition Domain Science XL, by the Dutch Research Council

(NWO) under the grant https://doi.org/10.61686/FHYZO53064.

References

[1] June 2025 TOP500, 2025. URL: https://www.top500.org/lists/top500/2025/06/, Accessed: August 2025.

[2] NVIDIA Corporation, CUDA C Programming Guide, NVIDIA, 2024. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/, Accessed: August 2025.

[3] Khronos Group, The OpenCL C Specification, Khronos Group, 2025. URL: https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_C.html, Accessed: August 2025.

[4] OpenMP Architecture Review Board, OpenMP Application Programming Interface, OpenMP ARB, 2024. URL: https://www.openmp.org/specifications/, Accessed: August 2025.

[5] S. K. Lam, A. Pitrou, S. Seibert, Numba: a LLVM-based Python JIT compiler, in: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM '15, Association for Computing Machinery, New York, NY, USA, 2015. URL: https://doi.org/10.1145/2833157.2833162. doi:10.1145/2833157.2833162.

[6] NVIDIA CUDA Compiler Driver 13.0 documentation, 2025. URL: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/, Accessed: August 2025.

[7] S. Klabnik, C. Nichols, The Rust Programming Language, 2nd Edition, No Starch Press, 2023. URL: https://doc.rust-lang.org/book/.

[8] crates.io: Rust package registry, 2025. URL: https://crates.io/, Accessed: August 2025.

[9] E. Holk, M. Pathirage, A. Chauhan, A. Lumsdaine, N. D. Matsakis, GPU programming in Rust: Implementing high-level abstractions in a systems-level language, in: 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, 2013, pp. 315–324. doi:10.1109/IPDPSW.2013.173.

[10] Rust-GPU, Embark Studios, rust-gpu: Rust as a first-class language and ecosystem for GPU graphics & compute shaders, https://github.com/Rust-GPU/rust-gpu, 2020-2025. Accessed: September 2025.

[11] Rust-CUDA: Ecosystem of libraries and tools for writing and executing fast GPU code fully in Rust, https://github.com/Rust-GPU/Rust-CUDA, 2021-2025. Accessed: September 2025.

[12] L. Faé, D. Griebler, Towards gpu parallelism abstractions in rust: A case study with linear pipelines, in: Anais do XXIX Simpósio Brasileiro de Linguagens de Programação, SBC, Porto Alegre, RS, Brasil, 2025, pp. 75–83. URL: https://sol.sbc.org.br/index.php/sblp/article/view/36951. doi:10.5753/sblp.2025.13152.

[13] N. Aukes, Hybrid compilation between GPGPU and CPU targets for Rust, Thesis, University of Twente, Enschede, 2024. URL: https://purl.utwente.nl/essays/100981.

[14] C.-A. Begu, Enabling Idiomatic Rust for Hybrid CPU/GPU Programming, Thesis, University of Twente, Enschede, 2025. URL: https://purl.utwente.nl/essays/107371.

[15] The Rust Project Developers, The rustc Development Guide, https://rustc-dev-guide.rust-lang.org/, 2025. Accessed: August 2025.

[16] Vulkan® 1.4.326 - a specification, 2025. URL: https://registry.khronos.org/vulkan/specs/latest/html/vkspec.html, Accessed: September 2025.

[17] J. Kessenich, B. Ouriel, R. Krisch, Spir-v specification, Khronos Group 3 (2018) 17.

[18] C. Legnitto, Rust running on every gpu, 2025. URL: https://rust-gpu.github.io/blog/2025/07/25/rust-on-every-gpu.

The Sampling Threat when Mining Generalizable Inter-Library Usage Patterns

Yunior Pacheco Correa[1], Coen De Roover[1] and Johannes Härtel[2]

[1]Vrije Universiteit Brussel, Brussels, Belgium
[2]Vrije Universiteit Amsterdam, Amsterdam, Netherlands

Abstract

Tool support in software engineering often relies on relationships, regularities, patterns, or rules mined from other users' code. Examples include approaches to bug prediction, code recommendation, and code autocompletion. Mining is typically performed on samples of code rather than the entirety of available software projects. While sampling is crucial for scaling data analysis, it can affect the generalization of the mined patterns.

We observe that limiting the sample to a specific library may hinder the generalization of inter-library patterns, posing a threat to their use or interpretation. Using a simulation and a real case study, we show this threat for different sampling methods. Our simulation shows that only when sampling for the disjunction of both libraries involved in the implication of a pattern, the implication generalizes well. Additionally, we show that real empirical data sampled using the GitHub search API does not behave as expected from our simulation. This identifies a potential threat relevant for many studies that use the GitHub search API for studying inter-library patterns.

Keywords

Sampling, Usage Patterns, Inter-Library, Dataset, Data Mining

1. Introduction

Sampling is crucial in empirical research, including Empirical Software Engineering (ESE) and Mining Software Repositories (MSR) [1, 2]. In MSR and ESE, researchers often sample software projects from sources like GitHub and aim to generalize their findings to unseen software projects. For studies that mine library or framework (API) patterns, sampling is equally important. Researchers extract API usage patterns from code by sampling examples from existing API applications.

Nuryyev et al. [3] for instance, mined the sample of 533 repositories from GitHub for annotation usage rules and validated them by human experts. We can find a rule of the following form:

$$type(\text{javax.json.JsonString}) \rightarrow annotation(\text{org.eclipse.microprofile.jwt.Claim})$$

This rule can be interpreted as a logical or probabilistic relationship, indicating that when a method returns a JsonString, it should carry a Claim annotation. This statement crosses different libraries, making it an inter-library pattern. Since JavaX's JsonString may also appear in other contexts, unrelated to MicroProfile's Claim annotation, the rule is not logically true. However, it may hold probabilistically with a certain confidence.

Nuryyev et al. discovered this pattern with unexpectedly high confidence. The authors assume JavaX to be an integral part of the MicroProfile framework. This is not true and renders it a rule between different libraries. We call it an *inter-library pattern* instead of an *intra-library pattern*. From a sampling perspective, we noticed that this distinction can be crucial. This problem leads us to ask: Is the confidence of a rule computed on the sample the same as for the entire population? In short, can we generalize? Is there a difference between intra-library and inter-library patterns? More concretely, we define our **research question** as follows:
How do sampling methods influence the generalizability of mined inter-library usage patterns from client software projects?

The 24th Belgium-Netherlands Software Evolution Workshop 17–18 November 2025, Enschede, The Netherlands
ypacheco@vub.be (Y. P. Correa); coen.de.roover@vub.be (C. D. Roover); j.a.hartel@vu.nl (J. Härtel)
0000-0002-7849-7841 (Y. P. Correa); 0000-0002-1710-1268 (C. D. Roover); 0000-0002-7461-2320 (J. Härtel)


2. Research Method and Results

To answer the research question, this paper follows a *research method* that combines an *empirical study* and a *simulation study* that examines inter-library patterns mined on data using different practices: i) *random samples*, ii) *single library samples* that collect client projects that use a particular library, and iii) *co-used library samples* that collect client projects by analyzing different combinations of usage of two libraries.

In the simulation study, we analyze the *generalizability* in terms of *confidence* for both intra-library and inter-library patterns. In the empirical study, we focus on inter-library patterns. The empirical and simulation studies examine the impact of different sampling methods on the mined inter-library patterns, considering the degree of popularity of the libraries involved.

We present evidence that *confidence* of mined inter-library usage patterns differs depending on the sampling method. Thereby, patterns may or may not generalize. For the specific case of mining inter-library usage patterns, we present an empirical study showing that mining on data sampled from GitHub does not behave as expected based on a corresponding simulation. Specifically, two sampling methods that should theoretically yield the same results instead produce different outcomes. This discrepancy suggests that at least one of the sampling methods is unreliable, assuming our simulation assumptions hold. Our findings highlight that the GitHub search API operates more as a black box than previously anticipated.

We provide a replication package online[1]. The resulting insights of our study can directly be used by future studies that extract patterns from samples. Either they **improve their sampling method in a way that the mined patterns generalize better**, or they **improve their threats to validity section by discussing the limitations that we identified**.

3. Conclusions and Future Work

At its core, this study examines the generalizability of findings in a specific area of software engineering. Our insights have practical implications: they can inform the choice of sampling methods in future research, or be highlighted as potential threats to validity in similar studies.

Relying only on data from single-library sampling for inter-library patterns (without manual validation) can weaken the validity of results. Random sampling is better but often not practical for rare libraries. Combining data from multiple sources and considering alternative mirror datasets like GHTorrent is recommended when possible.

Researchers and practitioners must stay alert. Recognizing and documenting the limitations of data collection and the black box nature of the GitHub search API is important to ensure the validity of mining studies. Developers using mined usage patterns in tools should also be careful, especially when working with patterns involving less popular libraries.

Future work should conduct more experiments to confirm these findings in similar settings. It is important to address and resolve the issues identified in this study. Further research should develop methods that help ensure the generalizability of results when mining inter-library usage patterns. This includes providing more transparent indexing mechanisms as alternatives to GitHub.

References

[1] V. Cosentino, J. L. C. Izquierdo, J. Cabot, Findings from github: methods, datasets and limitations, in: MSR, ACM, 2016, pp. 137–141.

[2] O. Dabic, E. Aghajani, G. Bavota, Sampling projects in github for MSR studies, in: MSR, IEEE, 2021, pp. 560–564.

[3] B. Nuryyev, A. K. Jha, S. Nadi, Y. Chang, E. Jiang, V. Sundaresan, Mining Annotation Usage Rules: A Case Study with MicroProfile, in: ICSME, IEEE, 2022, pp. 553–562.

[1] https://doi.org/10.5281/zenodo.14841462

An Analysis of Code Clones in GitHub Actions Workflows

Guillaume Cardoen[1], Alexandre Decan[1,2] and Tom Mens[1]

[1]Software Engineering Lab, University of Mons, Belgium
[2]F.R.S.-FNRS Research Associate

Abstract

GitHub Actions is the built-in CI/CD service of GitHub. While it promotes the use of reusable components, copy-pasting from existing workflows remains a frequent practice, which may lead to code clones. This paper explores the occurrences of code clones inside workflows. We conduct a quantitative analysis of 352K+ code clones instances in a dataset of 117K+ active workflows across GitHub. We observe that most workflow files contain non trivial code clones, mainly at the level of workflow steps. This study characterises code clones in GitHub Actions workflows, hence constituting the basis for understanding the impact of code clones in workflows.

Collaborative software development is an essential practice for globally distributed project teams. It relies on social coding platforms such as GitHub, providing a multitude of collaboration tools such as version control, issue and pull request management, quality analysis, code reviewing and continuous integration/deployement (CI/CD). GitHub is the most popular social coding platform, with over 5.2 billion contributions by over 100 millions users worldwide to more than 518 million repositories according to GitHub's 2024 Octoverse report [1].

In November 2019 GitHub released GitHub Actions (abbrevatied to *GHA*) as its built-in CI/CD tool, becoming the dominant CI/CD service in GitHub repositories in less than 18 months [2]. The GHA service requires repository maintainers to define and store *workflows* as YAML files in their repositories. These workflows declare how to automate repetitive tasks (e.g., testing, building, issue triaging, quality analysis, deploying) in reaction to one or more events (e.g., a push to a branch, a new pull request, a fixed schedule). When such an event occurs, a runner executes the corresponding workflow.

Workflows may use *Actions*, one of GHA reusable components. Actions can be of multiple types. A JavaScript Action is a single JavaScript file that is executed during runtime. Composite Actions use a similar syntax to that of GHA workflow, and can be used to declare a sequence of steps in a single reusable component callable from other workflows. Finally, *reusable workflows* share the syntax of GHA workflow and allow maintainers to reuse complete jobs.

Despite these reuse mechanisms, creating new workflows based on existing ones (either from the same author or from someone else) is a common practice [3]. Copy-pasting from one's own workflow is a frequent reuse mechanism used by GHA's users. This copy-paste reuse tends to lead to duplicated fragments within and across workflow files.

Such duplicated code is commonly referred to as *code clones*. Code cloning has been, and remains to be, a very active topic of research [4]. A common definition of code clones are code fragments that are similar according to some similarity measure [4].

Depending on the reasons of code cloning, it may have positive effects or to the contrary, be detrimental to the overall quality and maintainability of a codebase [5]. From the problematic side, clones tend to be considered as bad smells needing refactoring [6]. Clones may result in bug propagation, inconsistent bug fixes, reduced readability, increased code size, and obscuring the origin of the code [5]. From the beneficial side, code clones can enhance code readability and robustness while reducing development time [5]. Moreover, in languages lacking robust reuse mechanisms, they may be the only viable option for extending or adding a functionality without reinventing the wheel.

As GHA is widely used [2] as part of the supply chain of many software projects, there is a need to

BENEVOL 2025: The 24th Belgium-Netherlands Software Evolution Workshop Enschede, 17-18 November 2025
guillaume.CARDOEN@umons.ac.be (G. Cardoen); alexandre.DECAN@umons.ac.be (A. Decan); tom.MENS@umons.ac.be (T. Mens)
0009-0005-2008-3565 (G. Cardoen); 0000-0002-5824-5823 (A. Decan); 0009-0005-2008-3565 (T. Mens)

understand how cloning impacts GHA workflow maintainability, especially since the GHA documentation advocates to avoid duplication by relying on reusable components such as Actions and reusable workflows. A first step in this direction is to quantify the prevalence of code clones in GHA workflows, and to which extent they could be avoided.

We therefore shed more light on the characteristics of code clones within and across workflows in GitHub repositories. More specifically, we present a quantitative analysis of 352K+ instances of code clones identified in a large dataset [7] of 117K+ recently active workflows in 31K+ GitHub repositories to answer four research questions:

RQ1: *How prevalent are code clones in workflows?* This question aims to establish to which extent code clones occur in GHA workflows. We observed that a majority of workflow files contain non trivial code clones.

RQ2: *Which parts of workflows are subject to code clones?* While code clones are present in a majority of workflows, it remains unclear where they are located. Finding their location is essential to understand if refactoring is needed and possible. We found that the duplication of one or multiple shell commands or calls to reusable components represents four out of five code clones.

RQ3: *How are code clones distributed across different scopes?* GHA allow to declare reusable components callable from multiple workflows in the same GitHub repository or across the same GitHub organisation. This question analyses the characteristics of code clones across these different scopes.

RQ4: *To which extent could code clones be avoided by using appropriate reuse mechanisms?* This research question explores the possibility of refactoring workflows to remove the detected code clones by using GHA's existing reuse mechanisms.

Overall, we found evidence of widespread occurence of code clones inside GHA workflows and studied the characteristics of such clones. This study paves the way for further studies of code clones in workflows, as well as the impact of such clones.

Acknowledgments

This research is supported by F.R.S.-FNRS research projects T.0149.22 , F.4515.23 and J.0147.24.

References

[1] GitHub, Octoverse report 2024: The state of open source software, https://octoverse.github.com/, 2024. [Accessed 23-09-2025].
[2] M. Golzadeh, A. Decan, T. Mens, On the rise and fall of CI services in GitHub, in: International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2022.
[3] H. Onsori Delicheh, G. Cardoen, A. Decan, T. Mens, Automation and reuse practices in github actions workflows: A practitioner's perspective, 2025. doi:10.5281/zenodo.15422635.
[4] M. Zakeri-Nasrabadi, S. Parsa, M. Ramezani, C. Roy, M. Ekhtiarzadeh, A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges, Journal of Systems and Software (2023) 111796.
[5] C. J. Kapser, M. W. Godfrey, "cloning considered harmful" considered harmful: patterns of cloning in software, Empirical Software Engineering 13 (2008) 645–692.
[6] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 2018.
[7] G. Cardoen, T. Mens, A. Decan, A dataset of GitHub Actions workflow histories, in: International Conference on Mining Software Repositories, ACM, 2024.

SHOW: A Method for Inferring Python Proficiency from Textbooks

Ruksit Rojpaisarnkit[1], Gregorio Robles[2], Jesus M. Gonzalez-Barahona[2], Kenichi Matsumoto[1] and Raula Gaikovina Kula[3]

[1]*Nara Institute of Science and Technology, Japan*
[2]*Universidad Rey Juan Carlos, Madrid, Spain*
[3]*The University of Osaka, Japan*

Abstract

The accurate measurement of developer proficiency is paramount for ensuring software quality, as it directly reflects an individual's capacity to comprehend and produce efficient, effective, and well-structured code. While various code-based approaches for proficiency assessment have been proposed, the underlying process of learning coding concepts remains complex and widely debated. This paper introduces a novel framework for determining code proficiency by leveraging textbooks as ground-truth learning aids. The framework employs two automated methods, Übersequence and Clustering, to achieve this goal. We conducted an empirical study using a dataset of 22 introductory Python textbooks and Python AST code constructs. This analysis covered a high 85.51% of Python code constructs. Our findings demonstrate a remarkably high similarity in the sequential introduction of these constructs across the textbooks, validating the use of textbooks for proficiency assessment. The resulting Übersequence successfully assigns proficiency levels to individual code constructs, while the Clustering method provides a complementary, structured grouping perspective. We conclude by illustrating the framework's practical utility and discussing future applications in software maintenance tasks like bug assignment and code reviews.

Keywords

Software Maintenance, Software Evolution, Mining Software Repositories, Code Proficiency

1. Summary

In this talk –based on a research that has been accepted at TOSEM [1]– we introduce a novel, comprehensive framework for assessing code proficiency, explicitly designed to be programming-language agnostic. We validate its feasibility by applying it to the Python language, yielding promising results that successfully delineate distinct and meaningful groupings of proficiency levels. This achievement demonstrates that the complex challenge of accurately determining coding proficiency is both achievable and highly practical, validating the framework's core utility and establishing a new avenue for research with far-reaching implications. For researchers, the framework offers a structured methodology for studying coding skills at scale. In industry, practitioners gain a foundation for tools that can evaluate team capabilities, identify specific skill gaps, or optimize software maintenance tasks by aligning them with developer expertise. Furthermore, educators and students can leverage it to tailor learning curricula and track individual progress. The integration of clustering and the Übersequence provides a novel methodology for systematically mapping language constructs to proficiency levels, inspiring future research into automated or AI-driven systems for skill assessment. We recognize that most constructs identified in this initial study belong to Python's standard library. Therefore, we hypothesize that by adjusting the ground-truth sources (e.g., using specific textbooks for domains or specialized PyPI libraries), the framework can be customized to create personalized proficiency profiles appropriate for specific roles or technologies. Future work will focus on exploring the generalizability of this framework across diverse programming languages and specialized domain applications.

BENEVOL'25: Proceedings of the 24th Belgium-Netherlands Software Evolution Workshop, 17–18 November 2025, Enschede, The Netherlands

rojpaisarnkit.ruksit.rn1@is.naist.jp (R. Rojpaisarnkit); grex@gsyc.urjc.es (G. Robles); jesus.gonzalez.barahona@urjc.es (J. M. Gonzalez-Barahona); matumoto@is.naist.jp (K. Matsumoto); raula-k@ist.osaka-u.ac.jp (R. G. Kula)

Acknowledgments

This work is supported by the JSPS KAKENHI Grant Number JP20H05706, JP24H00692 and the Spanish Ministry of Science, Innovation, and Universities under the Excellence Network AI4Software (Red2022-134647-T) and through the Dependentium project (PID2022-139551NB-I00).

Declaration on Generative AI

During the preparation of this work, the authorss used generative AI in order to: Grammar and spelling check. After using these services, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] R. Rojpaisarnkit, G. Robles, J. M. Gonzalez-Barahona, K. Matsumoto, R. G. Kula, Determining code proficiency levels from python textbooks, Transactions on Software Engineering and Methodology Accepted; pending publication (2026). URL: https://arxiv.org/abs/2408.02262.

BRIDGE: Building Reliable Interfaces for Developer Guidance and Exploration through Static Analysis and LLM Translation

Krishna Narasimhan[1], Mairieli Wessel[2]

[1]F1RE BV, The Netherlands
[2]Radboud University Nijmegen, The Netherlands

Abstract

Although LLMs are often applied to code-related tasks, they fail to represent the links between syntax and identifiers, limiting their ability to reason about program behaviour. Static analysis tools capture these relationships accurately but remain difficult to use due to specialised query languages and complex interfaces. We present BRIDGE, a system that applies LLMs to translation and delegates program analysis to static tools. BRIDGE translates natural language queries into formal analysis queries, executes them with established tools, and adapts its responses according to developer proficiency. A proof-of-concept built with open-source components shows that even small models can perform accurate translations when provided with clear specifications. It answers developer queries in under a second, correctly resolves syntactic relationships, and adapts explanations to different skill levels. This *disruptive ideas and visionary explorations paper* outlines the system's architecture and an evaluation plan assessing accuracy, performance, and practical utility.

Keywords

static analysis, large language models, code understanding, developer tools, program analysis

1. Introduction

Modern developers utilize large language models help with a large variety of programming tasks, including and not limited to understand aspects of the code base. But there is debate whether LLMs serve as good replacements for code understanding tools. For example, when a developer asks questions about the code like "*what happens when this condition is true?*", an LLM may not connect the conditional keyword to the variable being evaluated. This reflects a broader limitation: LLMs do not reliably capture def-use chains, control dependencies, or data flow relationships that determine program structure and behavior [1]. As a result, they cannot consistently trace control flow or explain how program state changes under specific conditions. This limitation has practical implications for developers who depend on accurate reasoning about program semantics. Without reliable semantic information, LLMs can return responses that appear convincing but are in fact incorrect. Prior work shows that developers with different levels of experience interact with code understanding tools in distinct ways [2].

Past work has shown that static analysis tools that operate on program representations such as Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs) can provide an array of soundness and safety guarantee and are reliable sources of truth about the program's behavior. These representations allow them to, for example, determine how variables influence conditions and how data propagates through code [3, 4]. However, effective use of these tools requires specialist knowledge, since users must write queries in Domain-Specific Languages (DSLs) and follow complex workflows.

In this paper, we present **BRIDGE** (**B**uilding **R**eliable **I**nterfaces for **D**eveloper **G**uidance and **E**xploration), a system that employs LLMs as a translation layer. Natural language queries are converted into formal static analysis queries, which are then executed by established analysis tools. The responses are adapted according to indicators of the developer's level of experience.

BENEVOL 2025: 24th Belgium-Netherlands Software Evolution Workshop, November 2025, Location TBD
krishna.nm86@gmail.com (K. Narasimhan); mairieli.wessel@ru.nl (M. Wessel)
0000-0001-8004-3470 (K. Narasimhan); 0000-0001-8619-726X (M. Wessel)

The design of BRIDGE is motivated by the observation that transformers were originally developed for translation tasks [5, 6]. Translation remains one of their most reliable capabilities. By restricting LLMs to this role and delegating program analysis to static tools, BRIDGE assigns each component a clear and complementary function.

Our key contributions are (1) a method that separates natural language processing from code analysis; (2) an adaptation mechanism that adjusts responses based on observable indicators of developer experience; and (3) a proof-of-concept implementation using open-source components.

2. Related Work

Static analysis tools such as CodeQL [4] and Joern [3] provide accurate reasoning about programs through ASTs, CFGs, and DFGs but require specialised query languages and tooling, which limits accessibility [7]. LLM-based approaches, in contrast, often fail to capture the syntactic–identifier relationships needed for program flow reasoning. Anand et al. [1] showed that code-LLMs only encode token-level relations and, paradoxically, larger models capture less structural information than smaller ones. While this weakens their use for program analysis, transformers remain strong at translation, the task they were originally designed for [5, 6].

Research on developer interaction highlights that tool effectiveness depends on user style and experience. Richards and Wessel [2] observed that adaptive tools can improve outcomes for some users but harm others, showing the need for careful response adaptation. Other attempts, such as MoCQ [8], use LLMs to generate vulnerability patterns but still depend on security experts and do not address the gap in supporting general developers.

In contrast, our work confines LLMs to translation and assigns program reasoning to static analysis tools. This division allows us to combine the strengths of both approaches: deterministic results from static analysis and flexible query translation from LLMs. By additionally adapting responses to indicators of developer proficiency, our system lowers the entry barrier for novices while still providing concise and precise outputs for experienced users.

3. The BRIDGE approach

3.1. Design principles

BRIDGE operates on three principles. It restricts LLMs to translation tasks rather than code reasoning, as studies show that large language models fail to capture the relationships required for program semantics [1]. It relies on static tools for program analysis, since they provide deterministic results grounded in formal semantics. Finally, it adapts responses to different levels of developer proficiency by adjusting the level of detail based on features of the query [2].

3.2. System architecture

Figure 1 illustrates how the system separates natural language processing from program analysis and response generation in five components we describe below [1].

1. Query translation layer. This layer processes the natural language input. For example, when asked "How does `input_value` affect `result`?" it first identifies the type of question, such as data flow or structural analysis. It then extracts the relevant entities, such as function or variable names, and generates a formal static analysis query, for instance `TRACE FLOW FROM input_value TO result`.

2. Static analysis backend. This is the analytical core of the system. It receives the formal query from the translation layer and executes it against a precise, graph-based representation of the source code. This backend functions like a specialized database for code, allowing it to traverse the program's

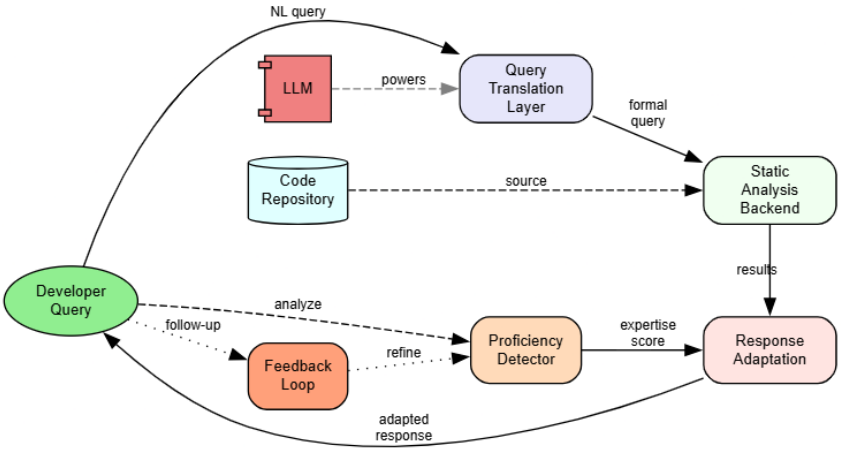


Figure 1: BRIDGE architecture with five components separating natural language processing, static analysis, and response adaptation.

structure and data flow paths deterministically. It returns structured, factual results, such as line numbers, call hierarchies, or the exact paths data travels between two points.

3. Developer proficiency detector. In parallel, a proficiency detector estimates the developer's skill level (i.e., expertise score E) from observable features of the query, including the specificity (the ratio of code identifiers to other words), the use of technical vocabulary, and whether the query is framed as a hypothesis (e.g., "Does X happen?") [2]. These factors are combined into a weighted score.

4. Response adaptation layer. This layer combines the results from the static analysis backend with the proficiency score (E). It then selects a response style suited to the user. For novices ($E < 0.3$), it provides step-by-step explanations and definitions of key concepts. For experts ($E \geq 0.7$), it returns a concise answer that highlights the main findings and possible edge cases.

5. Feedback loop manager. This component tracks conversational context, such as follow-up questions, to refine the expertise score over the course of a conversation session, making the system more adaptive over time.

3.3. Formal Properties

The static analysis backend of BRIDGE provides three formal properties: soundness, completeness, and determinism. It ensures **soundness**, meaning results are valid with respect to the semantics of static analysis. **Completeness** is defined relative to the capabilities of the underlying tools (e.g., Joern, CodeQL), so results include all behaviours that those tools can capture. The analysis is also **deterministic**, returning the same output for the same query on the same code. Note that these properties apply to the static analysis component; the LLM translation layer uses temperature=0 to minimize variability. These properties differentiate BRIDGE from approaches that rely only on LLMs. Detailed formulations are provided in Appendix A.

4. Implementation

We implemented a prototype of BRIDGE for Python code analysis, available on GitHub.[1] The system uses CodeT5-small [9], fine-tuned on 500 query–translation pairs. For static analysis, we leverage existing tools: Joern [3] for code property graphs and Tree-sitter [10] for parsing, with NetworkX [11]

[1] https://github.com/krinara86/Bridge-Benevol

for additional graph operations. A minimal domain-specific language (DSL) was defined to express queries such as `FIND USAGE OF x` and `TRACE FLOW FROM y TO z`, which map to graph traversals over abstract syntax trees and data flow graphs. In performance tests, response times were consistently under one second: query translation required 200–500 ms, static analysis 10–50 ms, and template-based response generation less than 10 ms. Further implementation details and examples are provided in Appendix B.

5. Planned evaluation

To assess BRIDGE, we propose an evaluation structured around four research questions (RQs). The study design combines quantitative benchmarks, correctness checks against established ground truth, and a user study.

RQ1: How effective is the natural language translation? We investigate whether natural language queries can be translated into formal analysis queries effectively. We plan to benchmark performance on a large dataset of query–code pairs collected from sources such as Stack Overflow and open-source projects. Translation quality will be assessed using standard machine translation metrics, including BLEU scores for n-gram similarity, together with semantic equivalence judged by two independent raters. These raters will determine whether the generated DSL query preserves the intent of the original natural language query. As a baseline, we will compare our fine-tuned CodeT5-small model with larger, general-purpose models such as GPT-4 and Llama 3 under zero-shot and few-shot prompting conditions. This allows us to test whether a specialised approach performs better than generic LLM prompting.

RQ2: Can BRIDGE correctly identify code dependencies where LLMs fail? To test this, we will construct a challenge set of queries that require tracing connections between variable definitions, conditional statements, and function calls (e.g., def-use chains, control dependencies). These queries will be executed both with BRIDGE and with a pure LLM baseline. These queries will be executed both with BRIDGE and with a pure LLM baseline. The outputs will then be compared against a ground truth established by manual static analysis. Following the methodology of Anand et al. [1], this comparison will reveal whether BRIDGE consistently returns correct results in cases where LLMs are known to fail by producing plausible but incorrect answers.

RQ3: Does expertise-based response adaptation improve the developer experience? This question evaluates the impact of the personalization layer. We will conduct a user study with more than 30 participants, divided into two groups: novices with less than two years of experience and experts with more than five years. Each participant will complete code understanding tasks using two versions of BRIDGE: one with adaptive responses and one with a fixed, intermediate-level response. The study will follow a within-subjects design to control for individual differences. Data will be collected using the System Usability Scale (SUS) to measure perceived usability, the NASA-TLX to assess cognitive load, and post-session questionnaires to gather feedback on clarity and usefulness. This will allow us to determine whether adaptive responses provide measurable benefits for novices and whether they affect the performance of experts.

RQ4: Does the system perform within interactive time limits? For BRIDGE to be a practical tool, it must provide responses quickly. We will measure end-to-end response times on Python projects of varying sizes (1k, 10k, and 100k lines of code), using a standardised setup such as an Apple M2 Pro with 16 GB RAM. Latency will be profiled for each component separately: query translation, static analysis, and response generation. The primary success criterion is maintaining a median response time of under one second across all query types, a threshold considered necessary for interactive use. This will help identify potential performance bottlenecks and determine whether the system can scale to larger projects.

6. Discussion

6.1. Design implications

Our design assigns translation tasks to LLMs and program reasoning to static tools. This division avoids known weaknesses of LLMs while taking advantage of their strength in translation. The same principle can be applied more broadly: statistical models are well suited for tasks such as translation or summarisation, while symbolic approaches are needed where correctness must be guaranteed. Personalisation in BRIDGE relies on observable features of the query, which makes the adaptation process explicit and easier to interpret compared to models that attempt to infer user intent [2]. We discuss a set of design implications and future work:

Query expressiveness: Complex developer questions may need to be decomposed into multiple sub-queries. For example, the request "Find unsanitised paths from input to database" must be divided into queries to identify inputs, database calls, sanitisation routines, and then paths between them. Future work may involve using an LLM to plan these decompositions automatically.

Cross-language support: Although the current prototype targets Python, extending the system requires support for multiple languages. A practical solution would be to use a shared intermediate representation such as a Code Property Graph, which combines abstract syntax, control flow, and data flow information, with only language-specific parsing needed.

Incremental analysis: Developers often require immediate feedback as they edit code, but re-analysing a large codebase on each change is infeasible. Efficient dependency tracking and caching strategies are needed to make analysis updates responsive.

Learning support for novices: Responses that only provide final answers without explanation risk reducing user understanding. For less experienced developers, outputs should include explanations of why relationships hold and how they can be identified.

Bias in adaptation: Since there is a reliance on training data, indicators of proficiency may correlate with communication style or background rather than skill. The system must be regularly audited for bias. Furthermore, developers should have agency over the system, with an option to manually set their preferred response style (e.g., "always give me the expert view") to override the detector.

6.2. Broader impact

Making static analysis more accessible can support projects and teams that do not have the resources for dedicated specialists. At the same time, it is important to state the limits of the system. BRIDGE can provide reliable results within the scope of static analysis but cannot replace human judgment for design choices, architecture, or requirement validation. The system is best understood as an assistive tool that extends developer capabilities rather than replacing them.

7. Conclusion

LLMs cannot reliably capture the syntactic–identifier relationships required for program reasoning, while static analysis tools provide accurate results but are difficult to use. BRIDGE combines these approaches by using LLMs for translation, static tools for analysis, and response templates that adapt to developer proficiency. Our prototype achieved 87% translation accuracy, resolved cases where LLMs alone failed, and produced results within one second. These results suggest that dividing tasks between neural and symbolic methods provides a practical way to build reliable developer tools.

References

[1] A. Anand, S. Verma, K. Narasimhan, M. Mezini, A critical study of what code-LLMs (do not) learn, in: Findings of the Association for Computational Linguistics: ACL 2024, Association for Computational Linguistics, Bangkok, Thailand, 2024, pp. 15869–15889.

[2] J. Richards, M. Wessel, What you need is what you get: Theory of mind for an llm-based code understanding assistant, in: 2024 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2024, pp. 666–671.
[3] F. Yamaguchi, N. Golde, D. Arp, K. Rieck, Modeling and discovering vulnerabilities with code property graphs, in: 2014 IEEE Symposium on Security and Privacy, IEEE, 2014, pp. 590–604.
[4] GitHub, Codeql: Semantic code analysis engine, 2024. URL: https://codeql.github.com/.
[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, in: Advances in neural information processing systems, 2017, pp. 5998–6008.
[6] J. Uszkoreit, Transformer: A novel neural network architecture for language understanding, 2017. URL: https://research.google/blog/transformer-a-novel-neural-network-architecture-for-language-understanding/.
[7] DeepSource, Globstar: The open-source static analysis toolkit, 2024. URL: https://globstar.dev/introduction.
[8] M. Team, Automated static vulnerability detection via a holistic neuro-symbolic approach, arXiv preprint arXiv:2504.16057 (2024).
[9] Y. Wang, W. Wang, S. Joty, S. C. Hoi, Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, in: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021, pp. 8696–8708.
[10] Tree-sitter developers, Tree-sitter: An incremental parsing library, 2024. URL: https://tree-sitter.github.io/tree-sitter/.
[11] A. A. Hagberg, D. A. Schult, P. J. Swart, Exploring network structure, dynamics, and function using networkx, in: Proceedings of the 7th Python in Science Conference, 2008, pp. 11–15.

A. Comparison of Approaches

Table 1
BRIDGE vs. Existing Approaches

Feature	BRIDGE	Pure LLM	CodeQL	Joern
Natural Language Input	✓	✓	×	×
Syntactic-ID Relations	✓	×	✓	✓
Deterministic Results	✓	×	✓	✓
No Query Language	✓	✓	×	×
Adaptive Responses	✓	Partial	×	×

*Within scope of analysis capabilities

B. Core Algorithms

B.1. Query Translation Algorithm

```
def translate_query(natural_query, llm, dsl_spec):
    # Step 1: Intent Classification
    intent_prompt = f"""
    Given query: {natural_query}
    Classify as: DATA_FLOW, CONTROL_FLOW,
                 STRUCTURAL, or DEPENDENCY
    DSL spec: {dsl_spec}
    """
    intent = llm.classify(intent_prompt)
```

```
    # Step 2: Entity Extraction
    entities = extract_code_entities(natural_query)
    entities += llm.extract_ambiguous_refs(
        natural_query, context=code_context)

    # Step 3: Query Generation
    if intent == "DATA_FLOW":
        if "from" in natural_query and "to" in natural_query:
            return f"TRACE FLOW FROM {entities[0]} TO {entities[1]}"
        elif "modifies" in natural_query:
            return f"WHAT MODIFIES {entities[0]}"
    elif intent == "CONTROL_FLOW":
        return f"PATHS TO {entities[0]}"
    # ... other intents

    return None  # Translation failed
```

B.2. Expertise Detection Algorithm

```
def compute_expertise(query_history, code):
    scores = []

    for query in query_history:
        # Specificity
        tokens = tokenize(query)
        identifiers = code.get_all_identifiers()
        S = len([t for t in tokens if t in identifiers]) / len(tokens)

        # Depth
        entities = extract_entities(query)
        if len(entities) >= 2:
            paths = [shortest_path(e1, e2) for e1, e2 in pairs(entities)]
            D = mean(paths) / graph_diameter(code.property_graph)
        else:
            D = 0

        # Hypothesis Ratio
        hypothesis_patterns = ["does", "is", "will", "could", "should"]
        H = 1.0 if any(p in query.lower() for p in hypothesis_patterns) else 0.0

        # Technical Vocabulary
        tech_terms = load_technical_vocabulary()
        T = sum(term_weight[t] for t in tokens if t in tech_terms) / len(tokens)

        # Weighted combination
        E = 0.25 * S + 0.25 * D + 0.25 * H + 0.25 * T
        scores.append(E)

    return mean(scores)  # Average over session
```

C. Case study: Complex query resolution

Consider a developer investigating a potential security vulnerability:

Natural language query: "Can user input reach the database query in process_order without going through sanitize_input?"

Step 1: Query decomposition BRIDGE decomposes this into sub-queries: (1) find all user input sources:`FIND USAGE OF request.params`; (2) find database queries: `FIND USAGE OF db.execute`; (3) find sanitization calls: `WHO CALLS sanitize_input`; and (4) compute paths avoiding sanitization

Step 2: Graph analysis

```
# Simplified path finding
user_inputs = {"request.params.order_id", "request.params.user_id"}
db_queries = {"line 45: db.execute(query)", "line 67: db.execute(update)"}
sanitization_points = {"line 23: sanitize_input(params)"}

vulnerable_paths = []
for input in user_inputs:
    for query in db_queries:
        paths = find_all_paths(input, query)
        for path in paths:
            if not any(san in path for san in sanitization_points):
                vulnerable_paths.append(path)
```

Step 3: Response adaptation

For a security expert (E=0.8): "Found 2 vulnerable paths: request.params.order_id → line 45 without sanitization. request.params.user_id → line 67 without sanitization. Both bypass sanitize_input through direct string concatenation."

For a novice (E=0.2): "I found a security issue! User input from web requests can reach your database without being cleaned first. This could allow SQL injection attacks. Here's what happens: 1. User data comes in through request.params 2. It gets used directly in database queries at lines 45 and 67 3. The sanitize_input function that should clean the data is never called Recommendation: Always pass user input through sanitize_input before using in queries."

D. Limitations and potential for future work

Dynamic code analysis: BRIDGE performs only static analysis. Dynamic features like reflection, `eval()`, or runtime code generation cannot be analyzed. This particularly affects languages like Python and JavaScript where dynamic behavior is common.

Inter-procedural analysis: Current implementation uses intra-procedural analysis. Calls across module boundaries or through function pointers may not be tracked accurately.

Context sensitivity: The prototype lacks context-sensitive analysis. Two calls to the same function are not distinguished, potentially leading to imprecise results for recursive functions.

Concurrency: Multi-threaded code with shared state presents challenges. Race conditions and synchronization issues require specialized analysis not currently implemented.

Training data: Fine-tuning requires manually created query-translation pairs. Automated generation of training data remains an open problem.

E. Extended implementation details

E.1. Graph construction details

Data flow graph construction:

```
def build_dfg(ast):
    dfg = nx.DiGraph()
    definitions = {}  # variable -> set of definition points
    uses = {}  # variable -> set of use points

```

```
    for node in ast_walk(ast):
        if is_assignment(node):
            var = get_lhs_variable(node)
            def_point = create_def_node(var, node.lineno)
            definitions[var].add(def_point)
            dfg.add_node(def_point)

            # Connect to previous definitions (kill-gen)
            for prev_def in reaching_definitions(var, node):
                dfg.add_edge(prev_def, def_point, type='kill')

        elif is_reference(node):
            var = get_referenced_variable(node)
            use_point = create_use_node(var, node.lineno)
            uses[var].add(use_point)
            dfg.add_node(use_point)

            # Connect to reaching definitions
            for def_point in reaching_definitions(var, node):
                dfg.add_edge(def_point, use_point, type='flow')

    return dfg
```

E.2. Response template system

Templates use a simple substitution system with conditional sections:

```
NOVICE_TEMPLATE = """
Let me explain what I found:

{?DEFINITIONS}
First, let me define some terms:
{DEFINITIONS}
{/DEFINITIONS}

Your question: {QUERY}

Here's what happens in the code:
{STEP_BY_STEP_EXPLANATION}

{?VISUAL}
Visual representation:
{ASCII_DIAGRAM}
{/VISUAL}

{?EXAMPLES}
Examples from your code:
{CODE_EXAMPLES}
{/EXAMPLES}
"""

EXPERT_TEMPLATE = """
{DIRECT_ANSWER}
{?EDGE_CASES}Edge cases: {EDGE_CASES}{/EDGE_CASES}
{?COMPLEXITY}Complexity: {COMPLEXITY_ANALYSIS}{/COMPLEXITY}
"""
```

On the Automation and Reuse Practices in GitHub Actions: Results of a Qualitative Survey

Hassan Onsori Delicheh[1], Guillaume Cardoen, Alexandre Decan and Tom Mens

[1]*Software Engineering Lab, University of Mons, Belgium*

Abstract

GitHub Actions is the dominant workflow automation tool for GitHub repositories. Workflow maintenance is often considered a burden for software developers, who frequently face difficulties in writing, testing, and debugging workflows. In the first half of 2025, we carried out an online survey of 419 GitHub workflow maintainers to understand their current automation and reuse practices, challenges, and preferences. We informed about the tasks that tend to be automated using GitHub Actions, the preferred workflow creation mechanisms, and the non-functional characteristics prioritised by respondents. We also examined the practices and challenges associated with GitHub's workflow reuse mechanisms. We observed significant disparities in automation adoption, with core CI/CD tasks being widely automated, but crucial areas like security analysis and performance monitoring receiving less attention. Next to GitHub Actions' built-in reuse mechanisms that are appreciated by many, we observed that copy-pasting remains a prevalent mechanism because it is perceived to be more convenient and allows for more control. These insights highlight opportunities for improved tooling, enhanced support for automation tasks, and better mechanisms for discovering, managing, and trusting reusable components.

Introduction

GitHub Actions is GitHub's integrated CI/CD mechanism. Since its release in 2019 it has become the *de facto* workflow automation tool for GitHub repositories. It enables repository maintainers to automate an unbounded range of automation tasks through workflow configurations. It also provides built-in support for reusable Actions and workflows, which can be shared and reused across workflows and repositories. Despite the widespread use of GitHub Actions, little is known about the automation and reuse practices adopted by workflow developers and maintainers. This hinders the development of best practices and tools to improve workflow automation for collaborative software development.

To fill this gap, we designed and conducted an online survey targeting GitHub workflow maintainers. Using a dataset of workflow commits [1] we identified 6,500 potential respondents having committed at least 10 workflow changes with at least one recent commit. We collected 419 complete responses from practitioners with demonstrated GitHub Actions experience. Data analysis comprised descriptive statistics, non-parametric hypothesis testing, and qualitative coding of free-text responses. The survey addressed two primary goals aimed at understanding GitHub Actions workflow usage and maintenance. Such understanding can ultimately lead to increased effectiveness of workflow usage and reduced maintenance effort.

G1: Understanding adopted workflow automation practices

To reach this goal, the survey included **three** questions related to the tasks being automated by workflows, the mechanisms employed to create these workflows, and the importance of specific non-functional characteristics during workflow maintenance.

1. Task automation. Testing, compiling and building dominate automation efforts, followed by code quality analysis and version management. Some critical tasks receive less attention: security analysis, performance monitoring and compliance checking. This suggests untapped automation potential.

BENEVOL 2025: The 24th Belgium-Netherlands Software Evolution Workshop Enschede, 17-18 November 2025

hassan.onsoridelicheh@umons.ac.be (H. Onsori Delicheh)

0009-0005-7935-4147 (H. Onsori Delicheh)

2. Workflow creation. Workflows are primarily created by adapting existing ones or writing them from scratch. GitHub's starter templates see more limited adoption, indicating a gap between available resources and practitioner needs.
3. Non-functional priorities. Reliability emerges as paramount, followed by security and understandability. Testing and debugging represent major pain points, with practitioners describing a trial-and-error approach due to inadequate tool support.

G2: Understanding workflow reuse practices

This goal comprised **four** survey questions to understand the motivations behind reuse in workflows, the characteristics that influence the selection and usage of reusable workflow components, and the challenges faced when incorporating them into workflows.
1. Reuse mechanisms. Actions developed by others are most frequently used, while reusable workflows see more limited adoption. Copy-pasting remains prevalent: maintainers frequently copy from own workflows, driven by convenience and desire for control.
2. Barriers to adoption. Key barriers include the difficulty to discover suitable Actions, complexity concerns, and limited awareness of the built-in mechanisms of composite Actions and reusable workflows. Trust issues are minimal despite security being a top concern.
3. Action selection criteria. When choosing Actions, respondents prioritise reliability, documentation, maintenance and security. With respect to license compatibility, respondents are uncertain of how this applies in the context of workflows.
4. Dependency Issues. Nearly all respondents encountered issues with Actions, most commonly because of outdated versions, deprecation and breaking changes. These challenges reinforce preferences for copy-pasting over external dependencies.

Conclusion

Our survey results provide solid empirical foundations for understanding GitHub Actions adoption patterns. We can derive several implications and recommendations from the survey results:
Practitioners could expand automation beyond core CI/CD to include security analysis, monitoring, and compliance checking. This could adopt robust dependency management practices and security tools to address Action-related vulnerabilities.
GitHub itself should enhance workflow testing and debugging support, improve template discoverability and customisation, and strengthen documentation for underused reuse mechanisms. Better search and filtering in the Marketplace could address discoverability issues.
Researchers should investigate the barriers to security automation adoption, develop metrics for assessing Action quality, and study long-term impacts of copy-pasting practices on workflow maintainability.

Our findings contribute to the growing body of knowledge on modern CI/CD practices and provide actionable insights for improving workflow automation tools and practices in collaborative software development environments.

Acknowledgments. This research has been submitted to ACM Transactions on Software Engineering and Methodology (TOSEM) and is currently under review. It is supported by the Fonds de la Recherche Scientifique - FNRS under grant numbers T.0149.22, F.4515.23 and J.0147.24.

References

[1] G. Cardoen, T. Mens, A. Decan, A dataset of GitHub Actions workflow histories, in: Int'l Conf. Mining Software Repositories (MSR), ACM, 2024, pp. 677–681. doi:10.1145/3643991.3644867.

On the Structuring of LaTeX Projects

Wouter ten Brinke[1], Bart Griepsma[1], Aleksandra Ignatovič[1], Nhat[2] and Vadim Zaytsev[2]

[1]Technical Computer Science, University of Twente, Enschede, The Netherlands
[2]Formal Methods & Tools (FMT), University of Twente, Enschede, The Netherlands

Abstract

In academia, LaTeX is a powerful typesetting system widely used for producing scientific documents such as research papers, theses and reports. It allows authors significant freedom and control over the structure and styling of their documents. However, this flexibility often leads to inconsistent internal project structures and coding styles, which can hinder maintainability and collaboration among co-authors.

In this paper, we investigate various existing traditions in structuring one's LaTeX projects. By analysing **29** academic users through interviews and surveys, we uncover prevalent practices and attitudes towards standardisation. Additionally, we mine **215** LaTeX repositories from GitHub to identify structural and stylistic patterns using feature extraction and clustering techniques. Finally, we introduce FlexiTeX, a system that allows users to maintain their preferred project structures while collaborating on shared content. FlexiTeX achieves this by parsing documents into an abstract tree representation and applying configurable transformation rules. Our preliminary findings suggest that while no universal standard exists, there is space for tool support in enhancing collaboration and maintainability in LaTeX projects.

1. Introduction

LaTeX [1] is a widely used typesetting system, particularly in academia, for producing high-quality scientific documents. Its strengths lie in its ability to handle complex formatting, mathematical notations, as well as bibliographies. LaTeX allows authors significant freedom in how they structure and organise their projects, and does not enforce any standards for folder layout, file naming conventions, coding styles, etc. Publishers often make use of their own *document classes* which impose some constraints on defining meta-information (authors' names, emails, title, subtitle, affiliations) and using certain packages, as well as *bibliography styles* which dictates which fields of BibTeX entries are used and how. A very occasional journal might employ a submission system that also limits font usage or requires all content to fit in one LaTeX file. Such unabashed flexibility can lead to inconsistent practices, making it challenging for collaborators to work together effectively, if they are used to drastically different folder structures or content clustering. Inconsistencies can also hinder maintainability, as authors may struggle in the future (when working on a resubmission, a camera ready version or an extended version of the same paper) to understand or modify documents that do not follow a clear and standardised format.

Despite its widespread use, there is currently no universally accepted standard for organising LaTeX projects. Authors often develop their own conventions for file structure, naming, and coding styles. These practices are often informal, ad hoc, and can vary widely across individuals and disciplines. This lack of standardisation leads to challenges in collaborative environments, where multiple authors may have different expectations and practices. In academia, such challenges are particularly pronounced, as scientific documents often involve multiple contributors(as is often the case for research papers) and require long-term maintenance activities (common for books and PhD theses). Services such as *Overleaf* aid collaboration by supporting various build configurations and providing templates, but they do not alleviate the issues one person's neatly curated setup is another's indecipherable labyrinth to navigate.

BENEVOL'25: Proceedings of the 24th Belgium-Netherlands Software Evolution Workshop, 17–18 November 2025, Enschede, The Netherlands

w.d.c.tenbrinke@student.utwente.nl (W. ten Brinke); b.griepsma@student.utwente.nl (B. Griepsma); a.ignatovic@student.utwente.nl (A. Ignatovič); research@nhat.run (Nhat); vadim@grammarware.net (V. Zaytsev)
https://grammarware.net/ (V. Zaytsev)
0009-0004-3110-9946 (Nhat); 0000-0001-7764-4224 (V. Zaytsev)

From functions and compile errors to version control and file structuring, LaTeX is closer to a *software project* than one might think. IDEs for it do exist, but less popular, more versatile and less standardised as in software development. In this project we explore how software engineering and evolution principles can be applied and adapted to the context of LaTeX writing. By treating LaTeX projects as software projects, we can leverage established software engineering and evolution practices to improve the experience and quality of LaTeX authoring.

In this specific work, we focus on project structuring and aim to answer the following questions:

RQ1 How do academics across different disciplines structure their LaTeX projects?
RQ2 What structural patterns can be observed in real-world LaTeX repositories?
RQ3 How can we support LaTeX collaboration without sacrificing existing personal project structures?

2. Related Work

Although LaTeX has become a standard tool in academic writing, there is surprisingly little research on how users organise their LaTeX project files or whether there are best practices. Most of the existing work focuses on teaching the basics of LaTeX, promoting templates, or improving user accessibility rather than directly studying file and folder structures.

Several researchers highlight the strengths of LaTeX and its widespread use in academia. Igel emphasises the steep learning curve of LaTeX, particularly when managing bibliographic styles and formatting requirements, but notes that its open-source flexibility makes it highly adaptable [2]. Zheng discusses the advantages of LaTeX over Microsoft Word in academic settings, describing how structured workshops help users become familiar with templates and tools [3]. Both sources suggest that while LaTeX enables standardisation in output, it offers little guidance for project organisation behind the scenes.

More technical efforts show how templates can reduce confusion and error. Frank et al developed a LaTeX-based reporting workflow using modular templates and automation scripts to support reproducibility in pharmacokinetic analysis [4]. Similarly, at Carnegie Mellon University, librarians used Overleaf to collaboratively redesign internal documentation, gaining insight into file management, templating, and project clarity [5]. These initiatives illustrate how structured LaTeX setups can benefit collaboration and efficiency, especially in institutional or team contexts.

Guizani and Rodríguez-Simmonds describe the role of student-led workshops in making LaTeX more accessible and community-oriented. Their work shows how inconsistent assumptions about file organisation can create confusion, even within a single department [6]. Meanwhile, Santos et al focus on academic libraries and propose template-based standardisation aligned with national formatting standards in Brazil, advocating for librarians to support LaTeX as a formal document preparation tool [7].

In general, while no studies have yet proposed a universal LaTeX file structure, several papers recognise the value of standardisation and reusable templates. These findings support the motivation for this paper: to investigate how academic users actually structure LaTeX projects and whether informal conventions could evolve into widely accepted standards. One of the few direct calls for LaTeX standardisation is Verna's article *Towards LaTeX Coding Standards* [8], which proposes a set of informal conventions based on programming best practices. Verna highlights the inconsistency of LaTeX source files and argues for clearer structuring, modularisation, and naming. However, his proposals are not based on empirical analysis, and no large-scale studies have tested whether LaTeX authors actually follow patterns that could support a shared standard.

Several tools exist that convert LaTeX documents into other formats. Pandoc [9] supports many input and output formats and can turn LaTeX into HTML, Markdown, or DOCX. LaTeXML [10], which is used by arXiv, focuses on preserving semantic structure when rendering documents as HTML [10]. plasTeX [11] parses and interprets macros to produce detailed transformations into formats like HTML. pylatexenc [12] provides utilities for parsing LaTeX code and converting it to Unicode. Although these tools effectively extract structured information, they are designed for converting documents into other formats rather than reorganising or reformatting LaTeX while staying in the same format. They also do not preserve all commands or macros, since many are unnecessary when targeting non-LaTeX outputs.

Some services like DBLP [13], CSAuthors [14] or BibSLEIGH [15] sanitise and standardise collections of BibTeX entries. However, they all do it in their own way, and the state of the art is that each experienced BibTeX user organises their own `.bib` files as they please, and commonly edits them manually to minimise dissatisfaction. The paper on BibSLEIGH highlights some problems of maintenance of BibTeX collections (such as links expiring with each redesign of publishers' websites), plus many issues in sanitisation of available data and metadata, such as using correct and distinct symbols for μ-kernel (microkernel, hence the micro sign, U+00B5) and μ-calculus (mu calculus, hence the Greek small letter mu, U+03BC), as well as opportunities in community analysis and bridging [16]. In this paper we intentionally focus on LaTeX and leave any related and unrelated BibTeX issues out of our scope.

3. Methodology and Contributions

This research project employs a mixed-methods approach, combining qualitative and quantitative techniques to explore LaTeX project structuring practices and develop a collaborative editing system. The contributions are divided into three main components:

- We conducted a qualitative study [17] involving semi-structured interviews and surveys with 29 academic users from various academic disciplines to understand their practices and attitudes towards LaTeX project structuring and standardisation. The study revealed dynamic cultural practices, shaped partly by syntax, and partly by the habits and preferences of individual members. Despite the diverging personal workflows, their responses converged on the need for a lightweight, flexible, modular, and community-driven framework that facilitates onboarding for newcomers and collaboration among co-authors.
- We performed a quantitative analysis [18] of 215 LaTeX repositories from *GitHub*, extracting features related to project structure and coding styles, and applying clustering techniques to identify common patterns with K-means as the clustering algorithm and Principal Component Analysis (PCA) for dimensionality reduction and visualisation. Structurally, the analysis revealed a wide range of practices, with no clear standard emerging. This enforces the lack of standardisation observed in the qualitative study. Stylistically, cluster variations suggested the absence of a common coding style, with differences in comment density, line lengths, and indentation styles.
- We introduced FLEXíTEX [19], a system that allows users to maintain their preferred project structures while collaborating on shared content. FLEXíTEX achieves this by parsing documents into an abstract tree representation and applying configurable transformation rules expressed in *YAML*. The transformation is designed to be reversible and idempotent, while preserving the ability to compile the document. Two collaborative workflows were proposed, one supporting turn-based collaboration and the other enabling mergeable collaboration when combined with a tool like `git diff`. A proof-of-concept was developed and demonstrated in a *GitHub* repository, showcasing the proposed workflows. Evaluation of this implementation on 324 real-world LaTeX projects from *GitHub* showed that FLEXíTEX performed well for projects with common macro usage and typical structure, but struggled with other cases due to parser limitations. Overall, FLEXíTEX demonstrates a promising approach to flexible LaTeX collaboration, though further work is needed to improve robustness and handle edge cases.

4. Results and Findings

4.1. RQ1: How do academics structure their projects?

All the statistics and findings are hard to fit in here, but we can include a few interesting points discussed during interviews we have conducted (number of interviewees included in brackets):

- Some (10) LaTeX users prefer to minimise folder usage or just use the root folder.
- Many (22) split their project in folders per section/chapter.

- Many have a dedicated folder for references (20), figures (22), code (9), tables (6), front matter (5).
- Half (15) of interviewed users stayed in one `main.tex` file without any modularity.
- Most popular file naming conventions are `snake_case` (7) and `CamelCase` (6), as well as numerical prefixes (14).
- When asked about strategies of separating content over files (e.g., file per section), no specific strategy (4) and a "mixed approach" (6) were surprisingly popular answers.
- The majority (19) supported possible future initiatives on standardising project structures.

4.2. RQ2: What structural patterns can be observed in repositories?

First, we collect relevant repositories from GitHub using its public API and our own bespoke script, expressing the following inclusion and exclusion criteria. We set the language filter to LaTeX and limit our search to repositories that included keywords such as "Thesis" and "PhD". To avoid test files and simple templates, we exclude any repositories with keywords like "template", "example", "sample", or "class" in their titles or descriptions. This step helps us narrow the focus to projects that reflect real-world usage of LaTeX for academic writing, rather than generic or instructional codebases. Then, we enforce the minimum repository size to 1MB, which serves as a rough indicator of content richness and helps screen out projects that are either incomplete or too small to provide meaningful structural insights. We clone all repositories selected this way, to make sure the complete folder structure and all associated files are available for processing. Our dataset is publicly available to enable replications [20].

After light screening of projects with unusual patterns and outliers, we pass them to the automated feature extraction script. We extract features by basically counting everything we can think of: number of LaTeX files, of `Makefiles`, of lines per file, of characters per line, of `\input/\include` commands, on average and maximum, etc. Then we use a combination of K-Means clustering to reveal latent structures in the data, and Principal Component Analysis (PCA) to reduce dimensionality. This yields four clusters (Figure 1, left):

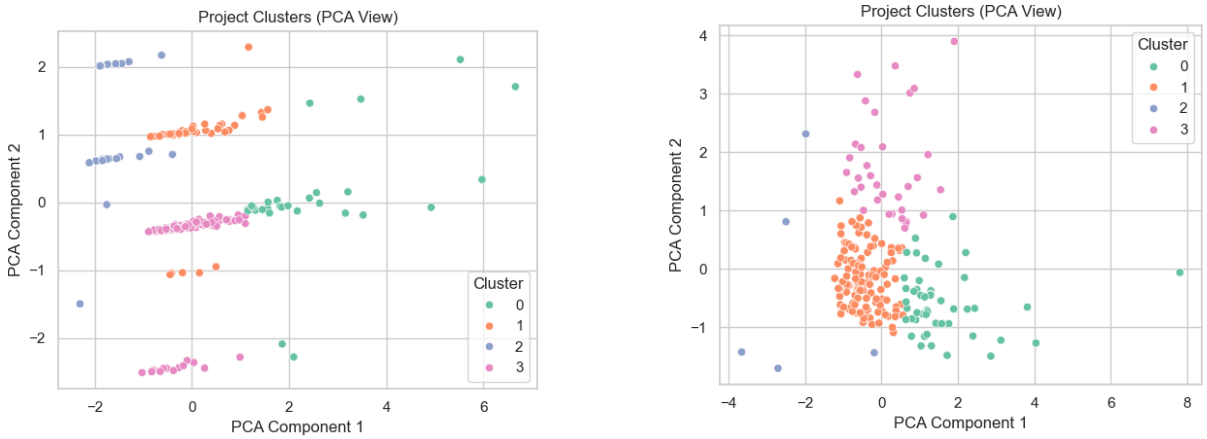


Figure 1: Clustering based on structural (left) and stylistic (right) features.

Cluster 0 stands out for its scale and complexity. Projects in this group often comprise dozens of `.tex` files spread across deep folder hierarchies, with total line counts exceeding 13000. Inclusion commands are consistently present in these repositories. In contrast, **Cluster 2** represents projects on the opposite end of the spectrum: comparatively small, averaging fewer than five `.tex` files and around 2200 lines, with shallow folder structures. *None* of the projects in Cluster 2 use inclusion commands. Between these two extremes lie Clusters 1 and 3, both of which reflect moderate project scales. **Cluster 1** projects typically contain around 21 `.tex` files and about four folders, accompanied by frequent use of inclusion commands. This group also shows a higher prevalence of Makefiles and README files. **Cluster 3**, while similar in size to Cluster 1, includes slightly fewer folders and projects, and rarely

uses Makefiles, although README files remain common. Clusters 0–2 have 30–40 projects each, while Cluster 3 has over 100 projects.

We repeat the same process focusing on different features, leading to different clusters. For instance, Figure 1, right, shows clusters based on features of readability and style. **Cluster 0** includes projects with very long lines (up to 2600 symbols), which indicates either generated content or tool support with soft newlines (like Overleaf offers). **Cluster 1** has much shorter lines (around 650 symbols) and even lower comment ratio. **Cluster 2** has lines of around 300 symbols maximum and just 24 on average, and tend to use tabs for indentation. **Cluster 3** has consistent line lengths (about 57 both for maximum and average) and has around 20% of lines in the project commented out as opposed to 5–6% in other clusters.

4.3. RQ3: How can we support LaTeX collaboration?

If Alice and Bob have each their own style of project organisation, but want to collaborate, this can be a case of applying bidirectional transformations (bx) [21] techniques, in particular forming a network of interacting bx [22]. That way, the `main` branch can host something in a "common style" with enough information to accommodate both Alice's and Bob's needs, who continue to work on their branches. Just like often the case with bx [23, 24], one can design a system based on states (*turn-based* on Figure 2) or on changes (*diff-based* on Figure 2).

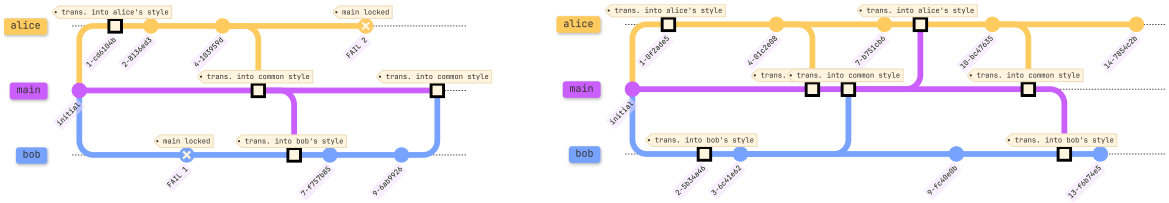


Figure 2: Proposed setups with FLEXiTEX: (left) turn-based, (right) diff-based.

We are currently conducting more experiments with FLEXiTEX [25], which is our prototype tool. It takes a configuration file specifying desired folder structure and conditions for content splitting, and is supposed to be run either locally as a git hook or remotely as an action. Preliminary results indicate that it can handle many real projects, but in particular tracking paths to files which are referred to through bespoke LaTeX commands, remains a challenge.

5. Concluding Remarks

In this work we explored how project structuring practices in LaTeX can be understood and improved through the lens of software engineering and evolution using a mixed-methods approach. Both the qualitative and quantitative analyses revealed a lack of universal standards, but also highlighted emerging informal conventions that could inform the development of flexible, community-driven guidelines. Building on these insights, we introduced FLEXiTEX, a system that enables users to maintain their preferred project structures while collaborating on shared content. For more information about the parts of this project, we advise consulting corresponding Bachelor theses that formed the core of this research [17, 18, 19].

Declaration on Generative AI

The authors have not employed any Generative AI tools to create, change or rephrase the content of this document.

References

[1] D. Knuth, L. Lamport, et al., LaTeX — A Document Preparation System, https://www.latex-project.org, 1984.

[2] C. Igel, Academic Writing with LaTeX, 2019. doi:10.22541/au.156080179.95968195.

[3] Y. Zheng, Academic Writing by Using LaTeX: A Hands-on Workshop, in: Proceedings of the 24th Annual Conference on Information Technology Education, SIGITE, ACM, New York, NY, USA, 2023, p. 90–91. doi:10.1145/3585059.3611425.

[4] T. Frank, S. Gastine, K. Lindauer, H. Speth, A. Strougo, A. Kovar, LaTeX Tutorial for the Standardization and Automation of Population Analysis Reports, CPT: Pharmacometrics & Systems Pharmacology 10 (2021) 1310–1322. doi:https://doi.org/10.1002/psp4.12705.

[5] H. C. Gunderman, D. Scherer, K. Behrman, Leveraging Library Technology Resources for Internal Projects, Outreach, and Engagement: A Case Study of Overleaf, LaTeX, and the KiltHub Institutional Repository Service at Carnegie Mellon University Libraries, College & Undergraduate Libraries 27 (2020) 164–175. doi:10.1080/10691316.2021.1885549.

[6] N. Guizani, H. E. Rodriguez-Simmonds, Developing Personal and Community Graduate Student Growth through the Implementation of a LaTeX Workshop, in: 2016 ASEE Annual Conference & Exposition, ASEE Conferences, New Orleans, Louisiana, 2016. doi:10.18260/p.26768.

[7] F. E. P. Santos, J. S. Lima, E. M. Rodrigues, I. L. d. Santos, K. Y. S. Feitosa, Desafios e possibilidades da atividade mediadora do bibliotecário na normalização de trabalhos acadêmicos: o uso do LaTeX, InCID: Revista de Ciência da Informação e Documentação 9 (2018) 25–51. doi:10.11606/issn.2178-2075.v9i1p25-51.

[8] D. Verna, Towards LaTeX Coding Standards, TUGboat 32 (2011) 309–328. URL: https://www.tug.org/TUGboat/tb32-3/tb102verna.pdf.

[9] J. MacFarlane, A. Krewinkel, J. Rosenthal, Pandoc, GPL-2.0 license, https://github.com/jgm/pandoc, 2006.

[10] B. R. Miller, D. Ginev, LaTeXML, National Institute of Standards and Technology, Public Domain license, https://math.nist.gov/~BMiller/LaTeXML/, 2004.

[11] K. Smith, PlasTeX, As-is license, https://github.com/plastex/plastex, 2007.

[12] P. Faist, Pylatexenc, MIT license, https://github.com/phfaist/pylatexenc, 2015.

[13] M. Ley, DBLP: Computer Science Bibliography, https://dblp.org, 1993.

[14] R. P. Barazzutti, CSAuthors.Net, https://www.csauthors.net, 2014.

[15] V. Zaytsev, BibSLEIGH, http://bibtex.github.io, 2015.

[16] V. Zaytsev, BibSLEIGH: Bibliography of Software (Language) Engineering in Generated Hypertext, in: A. H. Bagge, T. Mens, H. Osman (Eds.), Post-proceedings of the Eighth Seminar in Series on Advanced Techniques and Tools for Software Evolution (SATToSE 2015), volume 1820 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2017, pp. 54–64. URL: http://ceur-ws.org/Vol-1820/paper-06.pdf.

[17] A. Ignatovič, How Academics Organize LaTeX Projects — and Whether Structure Should Be Standardized, Bachelor's thesis, Universiteit Twente, Enschede, The Netherlands, 2025. URL: http://purl.utwente.nl/essays/107820.

[18] B. Griepsma, Can We Standardize LaTeX? Discovering Patterns in Real-World Repositories, Bachelor's thesis, Universiteit Twente, Enschede, The Netherlands, 2025. URL: http://purl.utwente.nl/essays/107264.

[19] W. ten Brinke, FLEXíTEX: LaTeX Collaboration Without Giving Up Personal Project Structure, Bachelor's thesis, Universiteit Twente, Enschede, The Netherlands, 2025. URL: http://purl.utwente.nl/essays/107262.

[20] B. Griepsma, LaTeX Academic Dataset, CC-0 license, https://github.com/Bart0TW/LaTeX_academic_dataset, 2025.

[21] K. Matsuda, R. Eramo, M. Johnson, V. Zaytsev (Eds.), Bidirectional Transformations — Foundations and Applications, National Institute of Informatics, 2025. URL: https://shonan.nii.ac.jp/seminars/231/.

[22] H. Giese, G. Karsai, V. Zaytsev, WG4: Multiple Interacting Bidirectional Transformations, in: A. Cleve, E. Kindler, P. Stevens, V. Zaytsev (Eds.), Report from Dagstuhl Seminar 18491 on Multidirectional Transformations and Synchronisations (MX Dagstuhl), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019, pp. 10–11.

[23] Z. Diskin, Y. Xiong, K. Czarnecki, From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case, Journal of Object Technology 10 (2011) 6: 1–25. doi:10.5381/JOT.2011.10.1.A6.

[24] Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, F. Orejas, From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case, in: J. Whittle, T. Clark, T. Kühne (Eds.), Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MoDELS), volume 6981 of *LNCS*, Springer, 2011, pp. 304–318. doi:10.1007/978-3-642-24485-8_22.

[25] W. ten Brinke, FLEXITEX, MIT License, https://github.com/wtb04/FlexiTeX, 2025.

ClassViz: From Inspection Tool to Research Vessel

Satrio Adi Rukmono[1]

[1]Eindhoven University of Technology (TU/e), De Zaale, Eindhoven, The Netherlands

Abstract

This work explores how visualisation bridges precise but complex software representations and the needs of human comprehension. We present *ClassViz*, a prototype that began as a lightweight inspection aid for code-to-graph instantiations and evolved through student projects, industrial collaborations, and research integration into a versatile platform for exploring software structures. Its trajectory illustrates how even a modest tool can become a shared environment for experimentation, evaluation, and communication.

Keywords

software architecture visualization, software maintenance, program comprehension

1. Introduction

Visualisation is central to understanding and maintaining software systems, complementing textual and structural representations. Graph-based models of source code are precise but often too large for direct inspection. In prior work, we proposed [1] labelled property graph (LPG)-based representations of software entities, relations, and attributes in a schema-light format [2]. Visualisation provides an intermediate form for inspecting instantiation outputs and communicating insights.

ClassViz[1] was developed to support such inspection, evolving into a platform used in research, education, and industry. It featured in the industrial evaluation of our deductive software architecture recovery (DSAR) technique [3, 4], acting as the "exoskeleton" for explanatory artefacts shown to participants.

2. Related Work

Software visualisation has addressed aspects of structure, behaviour, and evolution. A systematic review by Chotisarn et al. [5] of 105 articles (2013–2019) found that visualisations primarily target design, implementation, and maintenance tasks, typically using multivariate, graph-based, or metaphorical encodings (e.g., cities). Despite this diversity, industrial uptake remains limited, especially for maintenance and debugging. This underscores the need for practical and usable tools.

In software architecture visualisation, Shahin et al. [6] surveyed 53 studies (1999–2011). They identified four main types: graph-based, notation-based, matrix-based, and metaphor-based. Graph-based approaches dominate recovery and evolution tasks; metaphor-based views offer intuitive overviews, but raise scalability and cognitive concerns. Most techniques were tested only on small or academic systems, with little industrial use. The review emphasised the dual role of architecture visualisation: supporting both structural viewpoints (components, connectors, layers) and decisional viewpoints (design decisions and rationale).

These reviews frame our visualisation approach. ClassViz supports structural analysis of instantiated architecture graphs with filtering, layout, and layering. Its adoption in academic and industrial settings illustrates one response to calls for usable and practice-oriented tools.

BENEVOL'25: The 24th Belgium-Netherlands Software Evolution Workshop, November 17–18, 2025, Enschede, The Netherlands
s.a.rukmono@tue.nl (S. A. Rukmono)
https://satrio.rukmono.id/ (S. A. Rukmono)
0000-0001-9480-7216 (S. A. Rukmono)

[1]Live and loaded with JHotDraw 5.1 example input: https://satrio.rukmono.id/classviz/?p=jhotdraw-5.1

3. Visualisation Approach

ClassViz began as a pragmatic aid for visually inspecting code-to-graph instantiations [1]. It evolves through cycles of feedback, extension, and integration. Rather than a fixed set of requirements, its growth was driven by recurring needs arising from ongoing research. These needs form the basis for three research questions.

RQ1 What *visual affordances* support effective *lightweight inspection of labelled-property code graphs* for correctness and usability assessment?

RQ2 What *factors* influence the *adoption, extension, and appropriation of software structure visualisation tools* in educational and industrial contexts?

RQ3 How can software visualisation tools be designed to serve as *effective frontends* for diverse *automated analyses* such as architectural recovery and summarisation?

3.1. RQ1: Lightweight Visual Inspection of Graphs—Our Motivation and Origin

The initial version of ClassViz focused on inspecting large and complex graph instantiations produced by our tool Javapers[2]. These graphs were precise, but overwhelming, too large for immediate feedback. The initial design therefore emphasised clarity and filtering to produce diagrams that could be inspected during iterative development.

In ClassViz, nested boxes represented packages and classes, with class-level relations drawn as coloured UML-style line-with-end symbols (e.g., hollow triangles for inheritance, diamond for composition). Filtering allowed toggling of primitive types, packages, and relation types, and highlighting of classes by name. Click-through navigation revealed related elements, supporting localised exploration without overwhelming the view. Relations could be rendered as orthogonal or Bézier lines, with a choice of layouts from general-purpose algorithms. Figure 1 shows the initial version of ClassViz applied to JHotDraw 5.1.

These design choices enabled rapid inspection during the development of our LPG instantiation tool, allowing correctness and usability checks without heavy tooling. ClassViz offers minimal but expressive visual affordances—nested boxes, UML-style arrows, filters, and highlighting—that reduce cognitive load while preserving structural fidelity. Implemented as a lightweight browser application, it runs easily during short development cycles. **These simple encodings, filtering, navigation, and infrastructure made it an effective inspection aid, enabling quick structural sanity checks and supporting practical debugging of graph instantiation quality throughout development.**

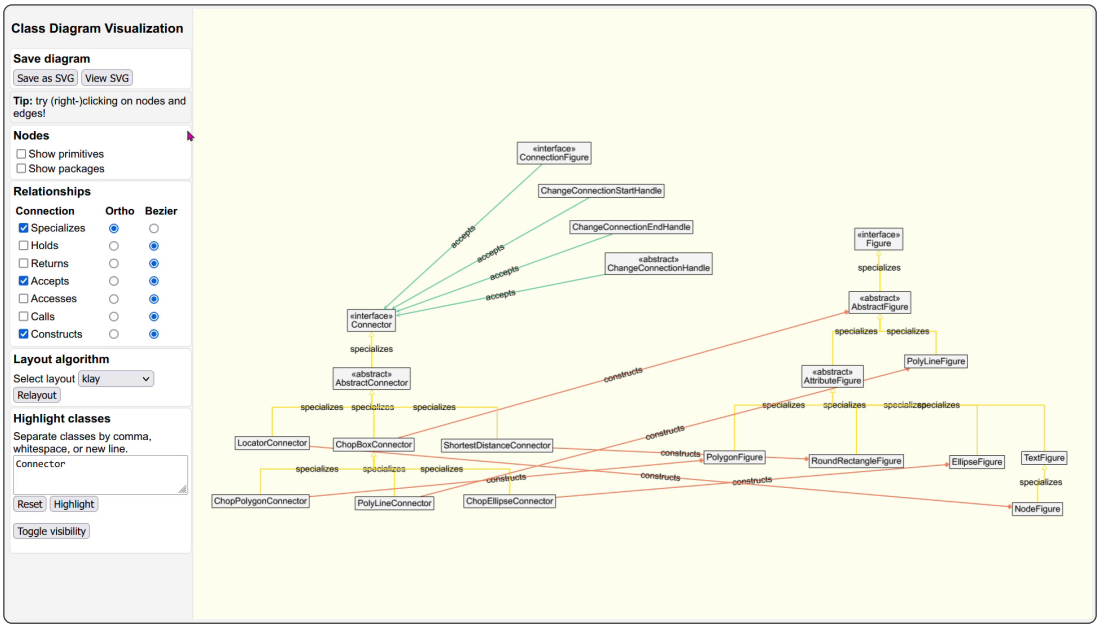
3.2. RQ2: Accessibility and Modifiability in Educational and Industrial Contexts

ClassViz's lightweight, browser-based architecture and minimal infrastructure assumptions made it easy to adopt and extend. Key factors that influenced its uptake in educational and industrial contexts include: (i) low barrier to entry for developers, (ii) openness to diverse input/output semantics in the same syntax, and (iii) modifiability by design. These qualities enabled it to function not just as a tool but as a shared platform across multiple independent student projects and industry-facing efforts.

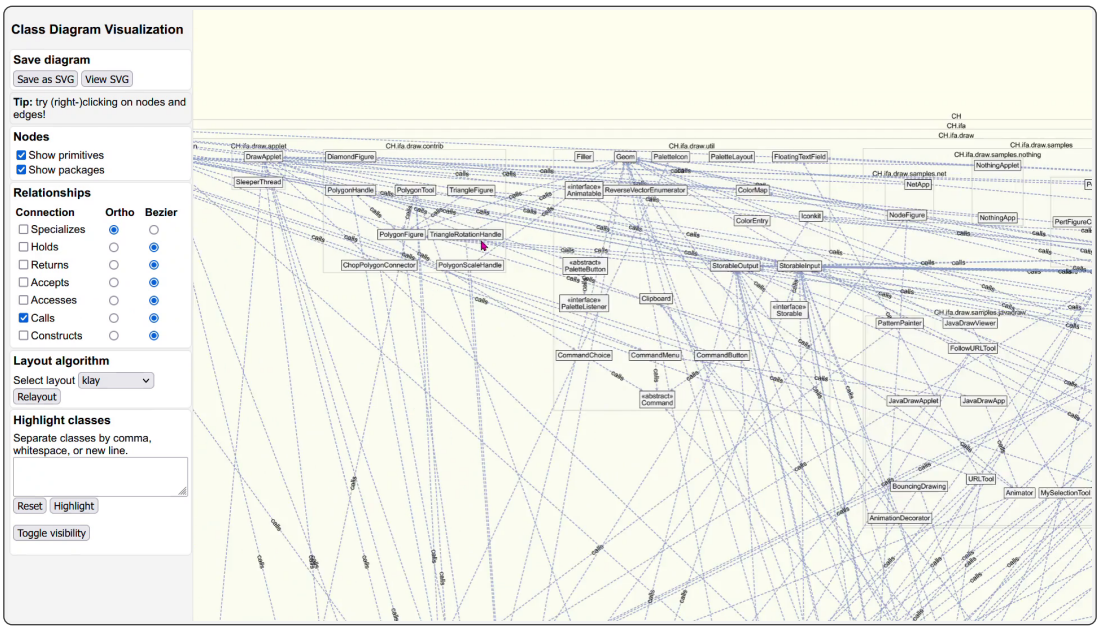
Several student extensions explored dynamic behaviour visualisation. Fung [7] added overlays for execution traces; Tanis [8] built a summarisation frontend for large-scale traces; He [9] combined static structure with activity overlays; and Van Esch [10] brought ClassViz concepts to virtual reality, combining structure and dynamic sequence data in 3D. Together, these works show how runtime-oriented adaptations were feasible across fidelity levels and modalities.

Other works focused on improving structural clarity and visual encoding. Kloet [11] addressed the legibility and usefulness of applying distinct layouts across abstraction levels. *BubbleTea* and *CodeView* [12, 13] introduced alternative layout metaphors: layered bubble-packing and abstracted

[2] https://github.com/rsatrioadi/javapers



(a) Classes involved in an instance of the strategy design pattern in JHotDraw 5.1. Other classes are filtered out.



(b) All packages and classes of JHotDraw 5.1 (that fit the viewport) and the “calls” relations among the classes.

Figure 1: Screenshots from an early ClassViz version, showing JHotDraw 5.1 in different modes.

views, respectively, while Atisomya [14] explored direct mapping of analytical dimensions to visual variables such as position and colour, showcasing the flexibility of the underlying design space.

ClassViz also served as a frontend for specialised analysis pipelines. Asuni [15] added overlays for vulnerability detection results. Kakkenberg et al. [16] applied ClassViz principles to low-code platforms, resulting in *Arvisan*, an industrially evaluated tool that continues to be useful in different domains [17].

Notably, the simplicity of ClassViz brings operational limitations that encouraged parallel evolution. Morier[3] developed a shared backend to support authentication, graph versioning, and permission-aware data access, laying the groundwork for a unified ecosystem across ClassViz forks and related tools.

Overall, these projects demonstrate that ClassViz’s adoption and extension were driven not by feature completeness but by a deliberately minimal and open design that enabled diverse appropriation.

[3]https://github.com/SimonMorier/ArchManager-back

The low entry barrier and minimal coupling empowered non-core developers to repurpose ClassViz across adjacent (sub)domains without deep reengineering.

3.3. RQ3: Visual Frontend for Architectural Analysis and Explanation

Feedback from students and collaborators led to iterative adaptations that broadened ClassViz's functionality and transformed it into a shared research artefact. A central enabler of this evolution is its use of LPGs, which allow analysis results to be integrated flexibly, either as additional nodes/edges or as properties on existing ones. These properties can be directly mapped to visual variables, e.g., metrics to colour gradients, classifications to discrete colours, and ranked values to spatial layout (e.g., vertical position by layer). This makes ClassViz well-suited as a frontend for structurally anchored explanations.

ClassViz was used as the presentation layer for hierarchical summarisation [18] and DSAR [3] outputs. To support these use cases, we added node colouring for classifications (i.e., role stereotypes [19], architectural layers [20]) and a detail pane for automatically generated summaries. These enhancements grounded abstract analysis results in the system's structure, improving interpretability and traceability.

Additional enhancements, some drawn from extension projects, were integrated to further support explanations. These include lifting and lowering relations across abstraction levels, gradient and thickness styling for edges to encode direction and weight, and more intuitive filtering and navigation. Figure 2 shows a recent version of ClassViz with role-classification results visualised using colour-coded nodes.

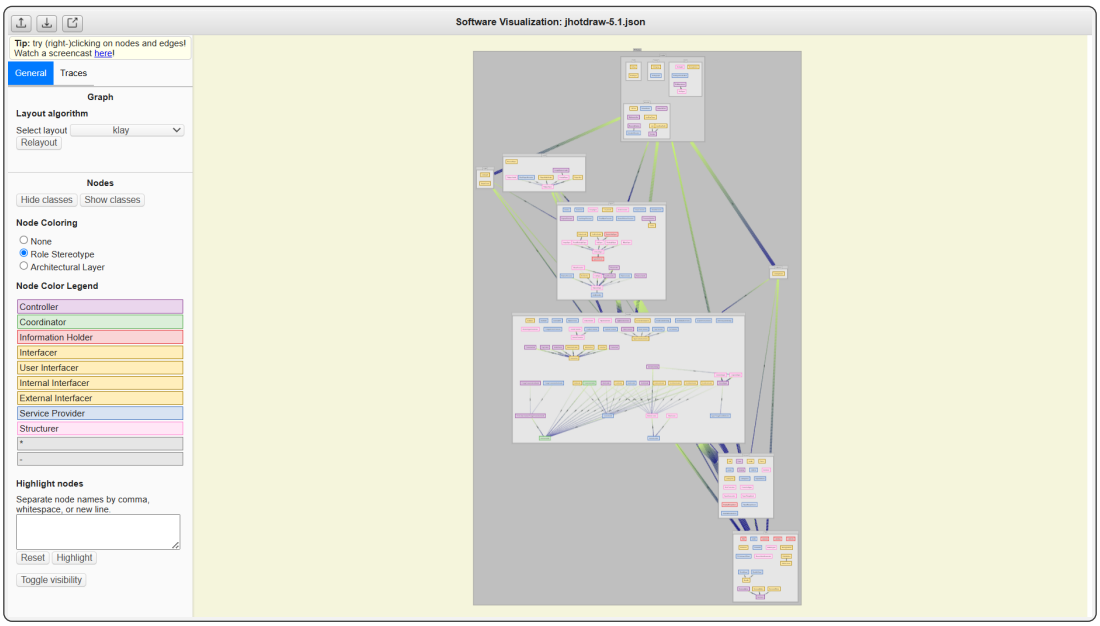


Figure 2: A recent version of ClassViz showing classes from JHotDraw 5.1 classified into role stereotypes [19] and call relation lifted into package level.

ClassViz was then deployed in an industrial evaluation at ASML. Our study [4], under review at ICSE-SEIP, assessed the tool in context: ClassViz acted as the explanatory surface for DSAR-derived architectural views. What began as a simple tool for visual inspection evolved into a presentation layer for explanation workflows, supporting interpretation and communication in both research and industry settings. **These integrations show how LPGs bridge abstract analysis results and architectural explanations by enabling node and edge properties to be directly mapped to visual variables.**

3.4. Eating Our Own Dog Food: Visualising ClassViz in ClassViz

Figure 3 shows the internal structure of the ClassViz source code, rendered in ClassViz itself with a manually arranged layout and minor post-processing for legibility. The diagram combines two abstraction levels: filesystem folders, shown in UML-style package notation; and JavaScript modules, shown as labelled rectangles. Modules are coloured by DSAR-inferred role stereotype (e.g., *Controller*, *Coordinator*); folders by architectural layer (i.e., *Presentation*, *Domain*, *Data*). Although ClassViz normally toggles between these modes, the figure overlays both to avoid duplication.

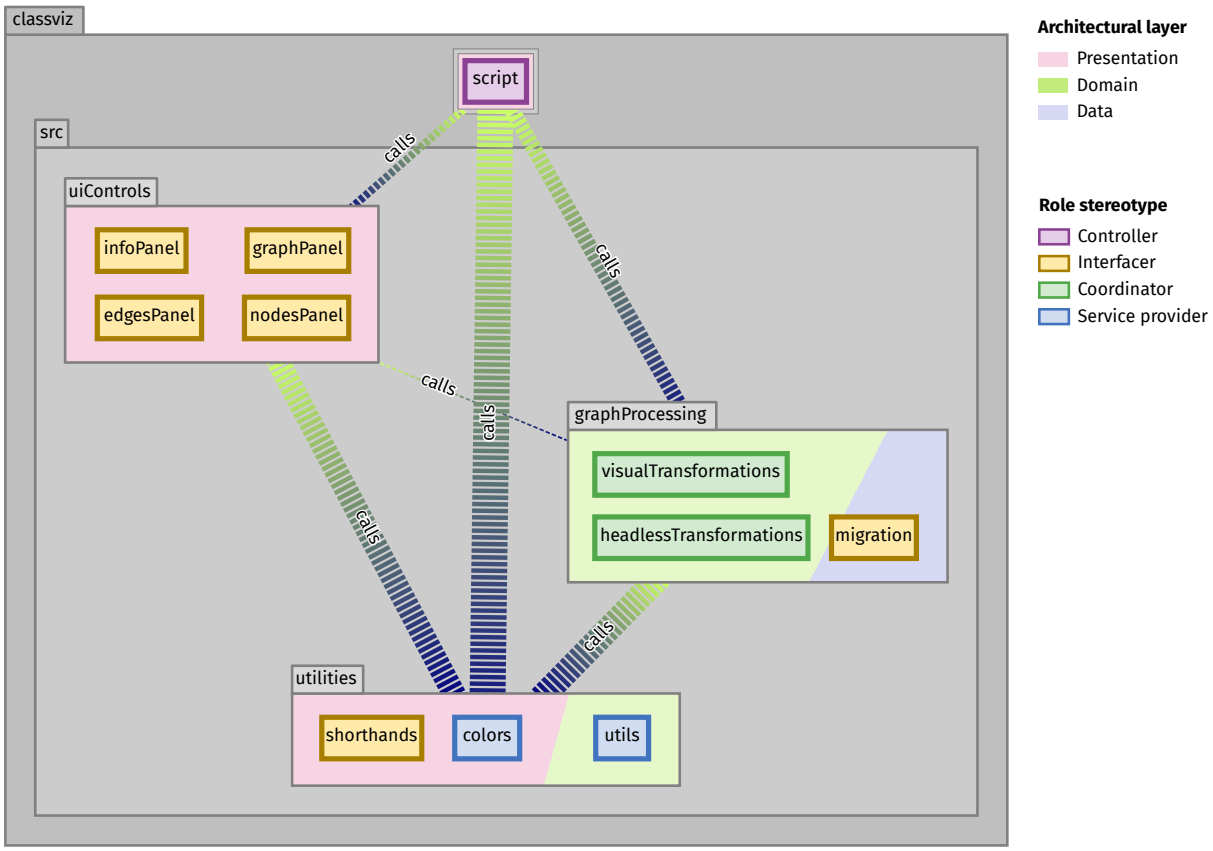


Figure 3: ClassViz source code organisation as depicted by ClassViz itself.

The `graphProcessing` folder implements the core pipeline, converting graphs into visual form by collapsing classes, assigning visual cues, and filtering edges. The `uiControls` folder handles user interaction, and `utilities` provides shared functions such as colour assignment.

This self-visualisation validates the tool on a non-trivial codebase and also offers design feedback: the dominance of calls to utility functions suggests overcentralisation, while the segmentation of UI panels highlights interface modularity.

4. Reflection

The trajectory of ClassViz shows how a pragmatic artefact can evolve into a central research instrument. Its growth was shaped by shifting research needs, with each inquiry prompting adaptations and, in turn, new questions. Visualisation tools in software engineering are rarely static; ClassViz illustrates how adaptability enables experimentation, pattern discovery, and explanation.

This adaptability stems from deliberate design minimalism and architectural openness. By avoiding rigid assumptions, ClassViz remained easy to extend, supporting overlays for dynamic behaviour, security, stereotypes, and VR with minimal friction. Its forks show that modifiability often outweighs

completeness. More broadly, explanation and inspection are not auxiliary but *generative*; seeing structure often precedes explaining it, making explanatory tooling a valid research contribution.

References

[1] S. A. Rukmono, M. R. Chaudron, Enabling Analysis and Reasoning on Software Systems through Knowledge Graph Representation, in: 20th International Conference on Mining Software Repositories, IEEE, 2023, pp. 120–124. doi:10.1109/MSR59073.2023.00029.

[2] D. Anikin, O. Borisenko, Y. Nedumov, Labeled Property Graphs: SQL or NoSQL?, in: 2019 Ivannikov Memorial Workshop (IVMEM), IEEE, 2019, pp. 7–13.

[3] S. A. Rukmono, L. Ochoa, M. R. Chaudron, Deductive Software Architecture Recovery via Chain-of-Thought Prompting, in: 44th International Conference on Software Engineering: New Ideas and Emerging Results, Association for Computing Machinery, New York, NY, USA, 2024, pp. 92–96. doi:10.1145/3639476.3639776.

[4] S. A. Rukmono, L. Ochoa, T. M. Bressers, J. Krüger, M. R. Chaudron, Evaluating Explanatory Artefacts of DSAR-Recovered Architectures from Industrial Codebases, 2025.

[5] N. Chotisarn, L. Merino, X. Zheng, S. Lonapalawong, T. Zhang, M. Xu, W. Chen, A Systematic Literature Review of Modern Software Visualization, Journal of Visualization 23 (2020) 539–558.

[6] M. Shahin, P. Liang, M. A. Babar, A Systematic Review of Software Architecture Visualization Techniques, Journal of Systems and Software 94 (2014) 161–185.

[7] K. Y. Fung, Classifying Java Classes Into Role Stereotypes Based on Their Behavior, Master's thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2023.

[8] H. Tanis, LLM-Enhanced Code Comprehension by Combining Static and Dynamic Analysis in Large-Scale C++ Systems, Master's thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2025.

[9] Y. He, Enhancing Program Comprehension through Visualization and LLM-based Summarization of Behaviour, Master's thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2025.

[10] H. v. Esch, Visualizing Software Behavior & Structure in Virtual Reality, Master's thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2024.

[11] P. Kloet, Multi-level Layout Algorithms for Visualizing Hierarchical Software Systems, Master's thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2024.

[12] S. A. Rukmono, M. R. Chaudron, C. Jeffrey, Layered BubbleTea Software Architecture Visualisation, in: Working Conference on Software Visualization, IEEE, 2024, pp. 122–126. doi:10.1109/VISSOFT64034.2024.00024.

[13] C. Jeffrey, A. P. Nugroho, S. A. Rukmono, Y. Widyani, CodeView: A Tool for Software Visualization in Development View, in: 2024 IEEE International Conference on Data and Software Engineering (ICoDSE), IEEE, 2024, pp. 67–72.

[14] A. K. Atisomya, Pengembangan Alat Visualisasi Arsitektur Perangkat Lunak untuk Analisis Multidimensi, Bachelor's thesis, Institut Teknologi Bandung, Sekolah Teknik Elektro dan Informatika, Bandung, Indonesia, 2025.

[15] M. Asuni, Effective Graphical Visualization of Vulnerabilities in C and C++ Programs, Master's thesis, University of Cagliari, Faculty of Engineering and Architecture, Cagliari, Italy, 2024.

[16] R. Kakkenberg, S. A. Rukmono, M. R. Chaudron, W. Gerholt, M. Pinto, C. R. de Oliveira, Arvisan: an Interactive Tool for Visualisation and Analysis of Low-Code Architecture Landscapes, in: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, 2024, pp. 848–855.

[17] F. Zamfirov, A. Radulescu, J. Krüger, M. R. Chaudron, Lessons from Visualizing Software Architecture Structure Conformance at Thermo Fisher Scientific, in: seaa, springer, 2025. doi:10.1007/978-3-032-04207-1_25.

[18] S. A. Rukmono, L. Ochoa, M. R. Chaudron, Achieving High-Level Software Component Summa-

rization via Hierarchical Chain-of-Thought Prompting and Static Code Analysis, in: International Conference on Data and Software Engineering, IEEE, 2023, pp. 7–12. doi:10.1109/ICoDSE59534.2023.10292037.

[19] R. Wirfs-Brock, A. McKean, I. Jacobson, J. Vlissides, Object Design: Roles, Responsibilities, and Collaborations, Pearson Education, 2002.

[20] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2012.

Pseudonymization as a Service: Compartmentalizing & Controlling Data Processing in Evolving Systems with Micropseudonymization

Job Doesburg[1,*], Bernard van Gastel[1] and Erik Poll[1]

[1]NOLAI, Radboud University, Erasmusplein 1, Nijmegen, The Netherlands

Abstract

IT systems can be complex, processing data for different purposes in different subsystems, e.g. in microservice or service-oriented architectures. With such growing complexity, the chance of compromise of one (sub)system increases. This comes with privacy risks. In this article, we demonstrate the strength of blind, polymorphic, transitive and distributed pseudonymization to cryptographically compartmentalize and control data, limit the impact of data breaches and preserve privacy during system evolution. While pseudonymization is a well-established technique to hide identities, we propose its application at much finer granularity to control data linkage between or even within microservices, which we call *micropseudonymization*. This extends functional compartmentalization to data compartmentalization, strengthening privacy. Specifically, we propose *Pseudonymization as a Service (PaaS)*: an architecture where different microservices process data about data subjects using different pseudonyms and communicate through a central pseudonymization service that monitors and controls data exchange. This allows for new subsystems to be added without (accidentally) violating privacy constraints, enabling privacy-preserving system evolution and implementing *privacy by design* by applying *pseudonymization by default*. While we explicitly propose a solution for microservice architectures, we believe our insight can be applied generally, for any data processing system with a functionally compartmentalized architecture.

Keywords

Compartmentalization, privacy, pseudonymization, microservices, privacy by design, software evolution

1. Introduction

Modern data processing systems can be complex. System architectures such as microservice or service-oriented architectures (SOAs), have enabled rapid growth of system functionalities. The rise of artificial intelligence, for example, drives the processing of data for additional purposes such as training or fine-tuning of AI models, and with the growing popularity of Software as a Service, an increasing number of legal parties are involved in data processing. Systems thus process data in an increasing number of subsystems, for different purposes, potentially by different parties (i.e. in different contexts [1]).

This growing complexity introduces privacy risks: the more subsystems are involved and thus the more complex the whole system grows, the higher the chance that eventually, one will be compromised, either intentionally (i.e. actors deliberately violating policies) or unintentionally (e.g. data leakage as result of hacking or failing security measures [2].). We see the consequences in numerous large-scale data breaches. To improve privacy without changing *what* data is processed (i.e. without processing *less* data), compartmentalization can be applied. Microservice architectures and SOAs are a form of *functional* compartmentalization, limiting the functional impact (i.e. the impact on the system's functioning, e.g. system integrity or availability) of compromise. The *data* they process, however, is often not actually compartmentalized due to the use of shared identifiers. Data can therefore be linked with other public data or data in other microservices. This may result in cascading privacy problems: data may appear in a different context [3] than intended (context collapse), and through combination of data (aggregation), new information may be revealed from patterns, exceeding the sum of parts [4], possibly even leading to (re)identification through quasi-identifiers [5].

BENEVOL'25: Belgium-Netherlands Software Evolution Workshop, November 17–18, 2025, Enschede, NL

*Corresponding author.

job.doesburg@ru.nl (J. Doesburg); bernard.vangastel@ru.nl (B. van Gastel); erik.poll@ru.nl (E. Poll)

0009-0004-4120-6977 (J. Doesburg); 0000-0002-0974-4634 (B. van Gastel); 0000-0003-4635-187X (E. Poll)

In this article, we demonstrate the strength of central, blind, polymorphic, transitive and distributed pseudonymization to compartmentalize data (in addition to functional compartmentalization), minimizing the privacy impact of (sub)system compromise, and enable privacy-preserving system evolution. While pseudonymization is a well-established technique to prevent identifiability of data subjects, we propose its application at much finer granularity to limit and control data linkage between different data compartments, formed by microservices. We call this *micropseudonymization*.

As we gradually explain in this article, this idea comes down to the following: where pseudonyms are normally only used to *hide* direct identifiers, micropseudonymization goes beyond this purpose and additionally aims to *separate* data processing in small isolated compartments, *minimize* the data processed in each compartment and *control* data linkage between them to centrally and cryptographically *enforce* specific privacy policies.[1] Or simply put: when each microservice uses its 'own' pseudonyms, unauthorized linkage of data across microservices is prevented by default, even when multiple services are compromised. This achieves significant privacy features, while being generic and flexible, which makes micropseudonymization a practical technology to implement: it only modifies the identifiers being used in communication between microservices. This maximally limits data linkability and performs data minimization without changing the functioning of the system. We consider this technology especially useful in complex systems with complex trust boundaries, e.g. involving different legal parties or actors in different roles that each may only see specific subsets of data.

Specifically, we propose *Pseudonymization as a Service (PaaS[2])*: an architecture where different microservices in a system processes data about the same data subjects using different domain-specific pseudonyms. Using a secure (i.e. blind, polymorphic, transitive and distributed) and central pseudonymization service, data can be converted and exchanged between these pseudonymization domains, while applying central monitoring and control on these conversions. The central authorities administering the pseudonymization service can enforce privacy policies that specify what data may be processed by which service. This helps to keep grip on data processing in complex and evolving systems and implements *privacy by design* by applying *pseudonymization by default*. We evaluate our ideas through a case study of a research data platform designed by the authors, that implements PaaS.

While the idea of using different pseudonyms for data in different domains to improve privacy is not groundbreaking and proposed earlier, for example for distributed governmental databases [7], doing it so granularly at the system architecture level and as a method to enable privacy-preserving system evolution in complex systems, has not yet been proposed to our current knowledge.

In order for micropseudonymization to be effective, we assume the data in each service to be strictly pseudonymous, meaning that they do not contain unique attributes (or combinations of them, called quasi-identifiers) shared with other services, that would allow for data linkage bypassing the pseudonymization service. Data should only be linkable via the identifiers. Applying micropseudonymization highly granularly, however, can actually be used to eliminate such quasi-identifiers in large parts of data processing, as we explain in subsection 3.5. We also do not consider timing and traffic analysis attacks, where data can be linked based on timing of events or other metadata. This typically holds for batch data processing in situations where the data subjects are not necessarily the 'users' of a system.

Terminology Throughout this article, we use the terms *system*, *subsystem*, *service* and *microservice* somewhat interchangeably for (software) systems that process data. Strictly speaking, a large data processing *system* can be composed of multiple *subsystems*, that all may also be considered as independent systems themselves, depending on the scope. For example, a large data processing system (perhaps even 'landscape') of a typical corporate organization may consist of different (sub)systems for customer relationship management, enterprise resource planning and billing, that all exchange data among each other. Some of these may be delivered *as a service* by a service provider. Others may be self-hosted and may even consist of more subsystems, such as a front-end, back-end and database service.

Typically, the term microservice is used to highlight the smallest self-contained unit of a system,

[1] Notice that *hide*, *separate*, *minimize*, *control* and *enforce* are all common privacy-design strategies [6].
[2] Not to confuse with *Platform as a Service*.

responsible for carrying out a single, well-defined data processing function, such as performing a specific computation, data transformation or doing 'data storage'. Systems following a microservice architecture consist of only such independent microservices, while SOAs typically group related data processing functions together in somewhat broader services. In this article, however, we do not focus on these differences and conveniently call each functionally compartmentalized unit of the system in scope a microservice, regardless of the size or complexity of the function they perform. This, for example, also includes externally delivered SaaS systems. Typical examples are separation in different software processes, in different Docker containers, on different physical machines, hosted by different legal parties or any other form of trust boundaries. Human actors manually processing data in some way could also be considered. We thus consider a model where a data processing system consists of a network of independent microservices, all performing a well-defined data processing function and exchanging data among each other. While we discuss microservices, our insights transfer to general data processing systems with various types of trust boundaries.

2. Background: identifiers and pseudonyms

We first provide a background on the usage of identifiers in data processing and different forms of pseudonymization (specifically PEP [8]), which forms the basis of our PaaS architecture.

Data processing systems rely on identifiers to organize and reference entities (or data subjects) within their data. Fundamentally, these identifiers serve two distinct functional purposes:

1. **Internal reference:** To distinguish entities *within* a specific dataset or system and *internally* link different data points or attributes relating to the same entity.
2. **External reference:** To refer to entities *across* different datasets or systems and *externally* link between representations of the same entity in multiple systems.

For internal reference purposes, the specific value of an identifier can be arbitrary, requiring only local uniqueness within the boundaries of that dataset or system, which we call the *domain*. For external reference, a higher degree of uniqueness is required to provide linkability across multiple domains. Notice that the definition of what is internal and external, depends on the scope at which we consider a system (see also section 1). For clarity, in the remainder of this article, we refer to *data* as the combination of *identifiers* (or *pseudonyms*), which have internal or external reference as their core purpose, and *attributes*, which are any data related to an identifier.

Identifiability A special case of external reference is (legal) identifiability, or the ability to reference natural persons, the strongest form of external reference.[3] This can be illustrated with a microservice architecture example. Internal reference applies within the boundaries of a single microservice. External reference applies between different microservices of the same system. However, not all microservices need to interface with data subjects (and hence identify them) directly. Microservices that do not interface with data subjects directly, but do need to communicate with each other, could thus use specific identifiers for external reference that do not need to be identifiable, i.e. pseudonyms. The definition of internal and external reference thus depends on the scope of and abstraction level at which we consider a system.

2.1. Pseudonymization

With pseudonymization, highly linkable, or even identifiable *identifiers* that allow for external reference are replaced by domain-specific *pseudonyms* with only limited linkability. Considering this has been done properly, i.e. there are no further linkable attributes in the data and pseudonyms are truly unlinkable,

[3] We follow the legal definition of identifiability. From a technical viewpoint, sometimes, any form of external reference is called identifiability, which is why these values are called *identifiers* (or *nyms*, from which *pseudonyms* are derived). We only use identifiability for direct reference to natural persons.

this prevents data in one *pseudonymization domain* from being linked with other data in other domains. Pseudonyms can only be linked across domains with knowledge of some secret *additional information* [9], i.e. the (reverse) pseudonym mapping. Linkability is thus reduced to secrecy of this information.

Notation A common way to denote a pseudonym for (id)entity $X \in \mathcal{I}$ in domain $A \in \mathcal{D}$ is $X_{@A}$ and the associated pseudonymization function for domain A, $P_A : \mathcal{I} \to A$, can be used to convert X into $P_A(X) = X_{@A}$. To recover a pseudonym, we can write $P_A^{-1}(X_{@A}) = X$.[4] However, for pseudonymization between different domains A and B as we consider in this article, we more generally denote $P : A \times \mathcal{D}_A \times \mathcal{D}_B \to B$ so that $P(X_{@A}, A, B) = X_{@B}$. Notably, in these cases for any entity x we have $P^{-1}(x, A, B) = P(x, B, A)$. We thus do not consider a distinct recovery function, but consider each identifier to be a pseudonym in its own domain. The initial value from which pseudonyms in other domains are derived will be called the 'origin'.

To denote that pseudonyms are truly unlinkable, we write $X_{@A} \perp X_{@B}$. More precisely, one could write $\perp_c$ to indicate computational unlinkability, for example based on the (decisional) discrete log problem ($\perp_{\text{DLP}}$), and $\perp_\infty$ to indicate information-theoretic security. For simplicity, however, we will not discuss these details and default to just $\perp$ in this article.

2.2. Properties of pseudonymization methods

There are various ways of computing pseudonyms, including randomized or counter-based mapping tables, cryptographic hash functions or Message Authentication Codes (MACs) and symmetric or asymmetric encryption [10]. They can have a variety of different properties of which we highlight the most interesting ones for privacy and security below.[5]

Randomness (pseudonym unlinkability) Generated pseudonyms are completely random and do not encode any other information, hence they are truly unlinkable. Counter-based pseudonyms, for example, violate this as they contain the order in which they are generated, which may reveal a relation, hence $X_{@A} \not\perp X_{@B}$.

Statelessness A pseudonymization method is stateless if its outcome does not depend on previously performed operations. This is relevant because stateless functions are typically easy to implement securely and scale well. While completely randomized mapping-table based pseudonymization can provide information-theoretic unlinkable pseudonyms (i.e. $X_{@A} \perp_\infty X_{@B}$), their internal mapping tables either need to be pre-generated (infeasible for large domains), or generated at run-time, resulting in a stateful pseudonymization function that does not fit the desired type (i.e. $P : A \times \mathcal{D}_A \times \mathcal{D}_B \times S \to B \times S$ instead of $P : A \times \mathcal{D}_A \times \mathcal{D}_B \to B$). Therefore, efficient stateless pseudonymization methods only offer computationally unlinkable pseudonyms.

Reversibility After pseudonymizing from domain A to domain B, it is possible to pseudonymize back to domain A (i.e. $P(P(X, A, B), B, A) = X$). Notably, we only consider *efficiently reversible* pseudonymization methods that require an inverse operation with the same efficiency as the regular operation. We thus exclude brute force approaches like *rainbow tables*, as these are generic attacks that are not specific to the pseudonymization method itself.

Transitivity Pseudonymizing from domain A to domain B and then from domain B to domain C without first reversing to domain A, yields the same result as pseudonymizing from domain A to domain C directly (i.e. $P(P(X, A, B), B, C) = P(X, A, C)$). Transitivity implies efficient reversibility.

[4] P^{-1} is sometimes denoted as recovery function R.

[5] For simplicity, we assume all pseudonymization methods to be *deterministic* and *collision-free*, though some non-deterministic methods with collisions do exist.

Distribution Pseudonymization is distributed when multiple parties need to be involved in the computation of a pseudonym. The benefit of distribution is that no single party possesses the actual pseudonym mapping and all parties need to be involved to convert a pseudonym. Consequently, the pseudonym mapping is less likely to leak while there is replicated monitoring and control. In fact, each pseudonymization method can be distributed by applying it multiple times in sequence, such as multi-tier mapping-tables where $P = P_2 \circ P_1$ (for 2 tiers).

Commutativity When pseudonymization is distributed over multiple parties (i.e. $P = P_2 \circ P_1$), the order in which the parties perform their operations does not matter (i.e. $P_2 \circ P_1 = P_1 \circ P_2$). Notice that, though every form of pseudonymization be distributed, only specific forms achieve commutativity. Commutativity allows for changing the order of parties, potentially limiting the linkability of intermediate values by these parties.

Blindness (homomorphism) Pseudonymization takes place blindly, thanks to homomorphic properties, without observing the identifiers that are being pseudonymized, i.e. pseudonyms are encrypted before they are sent through a pseudonymization process so the party performing pseudonymization does not observe the pseudonyms. The pseudonymization function $P = \mathsf{Dec} \circ P' \circ \mathsf{Enc}$ thus consists of an encryption and decryption function, as well as a homomorphic function P' on the encrypted pseudonyms.

Polymorphism (encryption randomness) Encryption is *polymorphic* (or randomized or probabilistic) if there are multiple unlinkable ciphertext representations of the same plaintext. For blind pseudonymization, this not only disables the party performing the actual pseudonymization from observing the pseudonym itself, but also whether it is pseudonymizing an entity it has already seen previously or not. This is sometimes also called multi-show unlinkability.

2.3. Encryption as pseudonymization method

From a cryptographic perspective, reversible pseudonymization can be considered as a form of deterministic (i.e. non-randomized) encryption of identifiers. The pseudonym mapping essentially forms the key used for encryption, called the *pseudonymization secret*. Vice versa, encryption methods can be used for pseudonymization, where the encryption key forms the *additional information*.

In contrast to how encryption is normally used, for pseudonymization, the decryption key should typically *not* be shared with the recipient, as this would allow them to undo the pseudonymization. Also, normally, ciphertext linkability is considered as an inherent vulnerability of deterministic encryption that do not use random values, nonces, salts, seeds or initial values. For pseudonymization, however, we deliberately use deterministic ciphertexts to form pseudonyms that link but hide the identifiers. Using different keys for different domains allows for pseudonym unlinkability between the different domains.

Blind pseudonymization can be considered as two-layer encryption, where the inner layer of encryption converts the origin identifier into a pseudonym (without the pseudonymizing party sharing the 'decryption key'), and the outer layer is applied to hide the origin identifier from the pseudonymizing party itself and securely send the pseudonym to the recipient (who *does* receive the decryption key). The first layer of encryption should be performed homomorphically and deterministically, resulting in the same 'ciphertext' every time. The second layer of encryption should be polymorphic.

2.4. PEP: Polymorphic Encryption and Pseudonymization

The PEP (Polymorphic Encryption and Pseudonymization) framework [8] is an advanced form of pseudonymization [10], that has all the properties of pseudonymization mentioned in subsection 2.2.[6] It uses homomorphic operations on malleable ElGamal ciphertexts to blindly convert encrypted pseudonyms

[6] Actually, some small extensions of the framework are required to securely implement transitivity and distribution, which are not discussed in this article.

from one domain into another, while using polymorphic encryption to make communication unlinkable for intermediate parties. The PEP framework has implementations in the 'PEP Responsible Data Sharing Repository' for medical research data [11], as well as the Dutch national eID system *DigiD Hoog* [12, 13].

Pseudonymization using PEP enables secure pseudonymization between many different domains, without introducing a privacy hotspot and without single points of failure. Additionally, PEP can be used for end-to-end encrypted *asynchronous* communication. This means that pseudonymization can happen on encrypted data, without the sender being involved (long after initial encryption), enabling encryption at rest. We do not discuss this further in this article.

3. Data compartmentalization through pseudonymization

Microservice architectures are a form of functional compartmentalization: services run isolated from each other in their own containers and are protected from interfering with each other. Therefore, when one service is compromised, this has only limited impact on the functioning of other services in the system. Regarding data, however, compromise of one compartment (e.g. data leakage in one service) does *not* necessarily have a limited privacy impact on data in other compartments: the impact of a data breach can extend beyond the loss of secrecy of only that data, through unauthorized data linkage with other public data or data in other services. As a result, data may appear in a different context than it was originally disclosed in, but more importantly, through combination (aggregation) with other data, new information may be highlighted, possibly even re-identifying otherwise anonymous data. Typically, there are no measures in place to maintain boundaries between compartments and prevent data linkage across them.

In this section, we first generally describe how data compartments can be formed, through the use of pseudonyms in different pseudonymization domains. We then discuss the usage of a central pseudonymization service to convert data between those domains, followed by the properties of pseudonymization methods required to do this securely. Finally, we describe how to apply this concept in microservice architectures, followed by more advanced use cases.

3.1. Defining data compartments

In order to implement actual data compartmentalization, data processing must be organized in such a way that compromise of data in one compartment does not have impact on data in other compartments. Therefore, linkability of data between different data compartments must be prevented. Specifically, distinguishing identifiers and attributes, this requires data compartments to be subsets of data with:

1. **unlinkable identifiers**: any identifiers (or pseudonyms) used for processing must be specific to that compartment and unlinkable to identifiers in other compartments.
2. **unlinkable attributes**: any attributes *shared* between compartments must be anonymous or unlinkable. Attributes or combinations of attributes that *are* linkable, may only be processed in a single compartment.

The challenge thus lies in smartly defining which subsets of attributes are processed together in individual microservices (see also subsection 3.4). We can then use pseudonymization to create unlinkable identifiers for the data subject for each of these subsets, to form compartments. Whenever data is exchanged between compartments, the identifiers are converted from the pseudonymization domain associated with the one compartment, to the pseudonymization domain of the other compartment.

Having defined the subsets of data that form compartments, we can describe what systems have access to that data compartment: all systems that are involved in the processing of that data in that specific pseudonymization domain. If any of these is compromised, we may need to consider the data in this data compartment to be compromised as well. This concept does not need to be limited to software systems, but could also include hardware, manual processes or people, similar to what is commonly called a Trusted Computing Base (TCB). We discuss this further in subsection 3.5.

Notice that regular (not blind) pseudonymization methods would violate our rules for data compartmentalization: the data compartment performing pseudonymization, contains linked identifiers in both pseudonymization domains. For now, we will briefly consider this as an exception and consider the pseudonymization process to be trusted. In subsection 3.3, however, we present how blind and polymorphic pseudonymization methods mitigate this.

Horizontal and vertical compartmentalization Notice that we specifically consider vertical compartmentalization, where different compartments contain different attributes (columns) about the same data subjects (rows). Strictly speaking, however, horizontal compartmentalization is also a form of compartmentalization, where different compartments consist of different sets of data subjects, or entries for data subjects, (rows) but with the same attributes (columns). There are, however, some interesting use cases for applying pseudonymization for horizontal compartmentalization. An architecture where one system processes data for one half of the data subjects, and another system processes data for the other half, for example, would be a form of horizontal compartmentalization. There is typically no pseudonymization required to enforce compartmentalization here, as there are typically no relations between different data subjects. A more advanced use case could be to record transaction logs of the same data subject under different pseudonym domains to break linkability.

3.2. Central pseudonymization as a service

A data processing system can only meaningfully function if data can be exchanged between its subsystems. To exchange data between two data compartments, a pseudonymization service is required, converting pseudonyms between two pseudonymization domains. When there are more than two domains between which pseudonyms need to be converted, there could be two distinct pseudonymization processes, one pseudonymizing from domain A to B, and another from domain B to C. This is, however, not a scalable solution as for n domains, $n-1$ pseudonymization processes are required of all processes need to communicate with each other.

An alternative to many different pseudonymization processes would be to centralize pseudonymization in a single service that performs all pseudonym conversions between all domains. This has a number of key benefits:

1. **pseudonymization key secrecy**: it is easier to properly secure a pseudonymization secret that is stored at a central place, than multiple pseudonymization secrets (or even multiple copies of the same secret) in multiple places.
2. **monitoring**: central pseudonymization is easy to monitor. Specifically, for example, brute force attacks that enumerate all pseudonyms are easier to detect and prevent.
3. **access control**: it is easy to enforce access control rules in a central place. This provides flexibility to change policies in a single place.

There is, however, one important reason why a central pseudonymization service is undesirable using regular pseudonymization methods: central pseudonymization forms a great privacy hotspot. A regular central pseudonymization process could observe all data being exchanged between different data compartments and would be a single point of failure for the confidentiality and unlinkability of data. This would effectively undo any positive privacy effects of data compartmentalization. Considering this central pseudonymization service as a trusted, as stated before, would be an unreasonable assumption.

3.3. Blind, polymorphic, transitive and distributed pseudonymization

Regular pseudonymization methods are a single point of failure for the confidentiality and unlinkability of the pseudonyms they convert. There are, however, pseudonymization methods that do not suffer from these problems. Specifically, pseudonymization methods that are *blind*, *polymorphic*, *transitive* and *distributed* do not form a single point of failure or privacy hotspot.

1. **blind**: First, *blind* pseudonymization methods do not suffer from the privacy hotspot problem, because data is encrypted before it is sent through the pseudonymization service. The pseudonymization service itself does not have access to any data and is thus not part of any data compartment itself. There is thus no privacy hotspot.
2. **polymorphic**: In particular, *polymorphic* blind pseudonymization is required to prevent the pseudonymization service from linking the same pseudonym over different conversions.
3. **transitive**: Pseudonymization should be *transitive* when there are many compartments and many data flows between them. Without transitivity, pseudonymization from domain A to C (or C to A) must always go through domain B, causing linkability there.
4. **distributed**: Finally, it is desirable to perform pseudonymization in a *distributed* way, i.e. by multiple parties to eliminate single points of failure. Even if pseudonymization is performed blindly, it is still based on a pseudonymization secret which can leak. If in a n-tier distributed pseudonymization process one party leaks their pseudonymization secret, this does not directly break pseudonym unlinkability. Only if all n parties would be compromised, pseudonym unlinkability breaks. Additionally, *commutativity* may be a desirable property for practical implementations, but it does not prevent any particular privacy or security threats. In the remainder of this article, for simplicity, we will discuss a single pseudonymization service, which internally could consist of multiple distributed subsystems that together form a single pseudonymization service.

As mentioned in subsection 2.4, the PEP pseudonymization has all these properties.

3.4. Micropseudonymization

The idea of micropseudonymization is to organize data processing in the smallest possible data compartments. Ideally, each data compartment precisely belongs to a single microservice, performing a single well-defined data processing function, and consists of precisely the data required for executing that function. Since each microservice performs a single function, they often process different sets of attributes as input or compute different attributes as output.[7] Therefore, each microservice could be considered to process its own subset of data and can form its own data compartment.

If this is the case, each compartment has proper data minimalization and the boundaries of our data compartments are perfectly aligned with the trust boundaries of the (functionally compartmentalized) microservices. This maximally limits the impact of compromise of each microservice.

Whenever data needs to be exchanged between microservices, this must go through the pseudonymization service. Transitivity guarantees that data from different microservices can be properly combined, but only if the pseudonymization service allows it according to its configured policies. This is especially useful if microservices request data *just in time* (only at the time they need it), as this enables the pseudonymization service to log whenever data was used by a specific microservice.

Toy example We consider a (highly simplified) toy example of an e-commerce system. This system consists of microservices for *user authentication*, *order fulfillment*, *payments* and *shipping*. Each microservice forms their own data compartment:

- The *user authentication* data consists of usernames and passwords, stored under a user ID in the user-authentication-domain U $uid_{@U}$.
- The *order fulfillment* data consists of ordered products and parcel numbers, stored under a user ID in the order-domain O $uid_{@O}$.
- The *payment* data consists of bank transactions and payment confirmations, stored under a user ID in the payment-domain P $uid_{@P}$.
- The *shipping* data consists of payment confirmations and parcel numbers[8], stored under a user ID in the shipping-domain S $uid_{@S}$.

[7] An exception to this could be databases, which we discuss in subsection 3.5.

[8] As explained later, this may actually be a violation of data compartmentalization if the parcel number is unique.

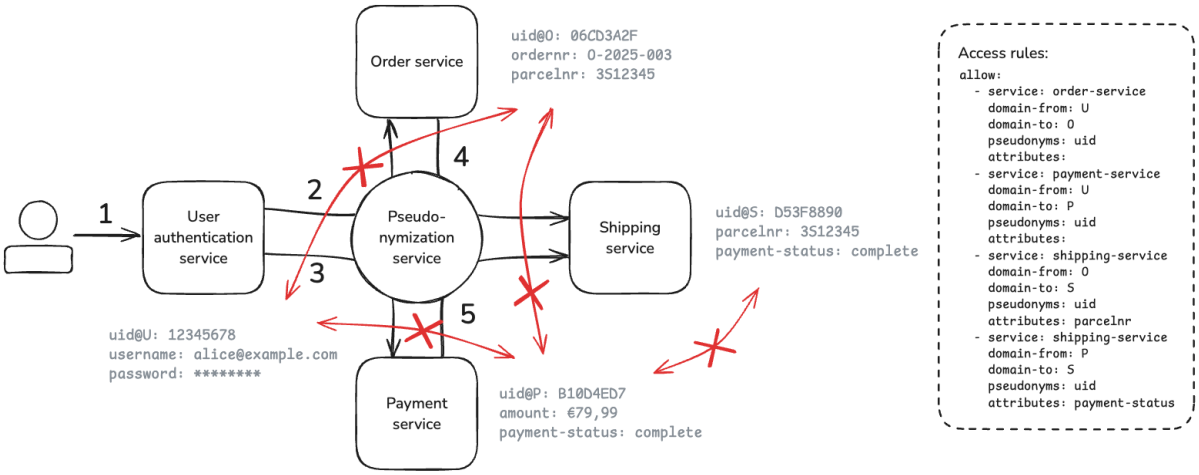


Figure 1: A toy example of an e-commerce system implementing PaaS, where data cannot be linked between the different microservices (in red) as $uid_{@U} \perp uid_{@O} \perp uid_{@P} \perp uid_{@S}$.

A typical flow in this system may look as follows (see also Figure 1).

1. To authenticate, user X provides their username and password to the user authentication service.
2. Upon placing an order, their associated user ID $X_{@U}$ is converted through the pseudonymization service and sent to the order service, who receives $X_{@O}$ to use in the fulfillment of the order.
3. Meanwhile, $X_{@U}$ is also sent to the payment service that uses $X_{@P}$ to process the user's payment.
4. After payment, the payment service sends a payment confirmation to the shipping service, converting $X_{@P}$ into $X_{@S}$.
5. The order service may send the parcel number to the shipping service, converting $X_{@O}$ into $X_{@S}$. The shipping service now has both the payment confirmation (from the payment service) and parcel number (from the order service) and knows it can ship the parcel.

Notice that except for the authorized data flows (see the access rules in Figure 1) that go through the pseudonymization service, data cannot be linked. The payment service does not know what was ordered, the order fulfillment service does not know anything about payments and no service except the user authentication service knows the user's identity. When the data in any of these microservices leaks, they remain unlinkable to other data in other microservices. An exception is the shipping service and the order service that share a parcel number and could combine their data, assuming that the shared parcel number is unique (which would indeed violate the unlinkable attributes assumption of section 3).

We could extend this scenario with a customer support service. This service could be allowed to request data from all different microservices, converting all those pseudonyms into a customer support pseudonym $uid_{@C}$. In order to allow this, however, the pseudonymization service could implement a policy where a customer support agent first needs to justify their request through a customer support ticket before pseudonymization is allowed to happen. This architecture can thus also be used to enforce more complex privacy policies, instead of just static ones.

3.5. Advanced use cases

So far, we considered data compartments to belong to a single microservice to minimize the amount of data processed in that compartment and limit the attack surface. The reverse, however, does not need to be true. There are actually good reasons for a single microservice to operate in multiple pseudonymization domains and multiple compartments (see also vertical compartmentalization in subsection 3.1).

For example, a survey system may process different surveys for the same set of respondents. When using a single pseudonymization domain, their responses to different surveys would be linkable within that survey system. By associating each survey with its own pseudonymization domain and applying

pseudonymization between them, linkability of surveys can be prevented. The system will thus process responses to survey 1 under $X_{@survey1}$ and to survey 2 under $X_{@survey2}$. For true unlinkability in case of compromise, it is important that the survey system itself is not allowed to convert pseudonyms between the different pseudonymization domains.

A similar construction could even be used in our e-commerce system to make different orders of the same customer unlinkable to each other within the order fulfillment service, with different pseudonymization domains per month, week or day (or even order number). By applying this thoroughly and at a fine-grained level, every individual attribute could be stored under a different pseudonym. Only when two attributes really need to be combined for a specific computation, their pseudonyms can be converted *just in time* by the pseudonymization service for the short duration of that computation. This way, even quasi-identifiers can be eliminated during significant parts of data processing.

Database example A concrete example of this concept could a stateful microservice that covers multiple trust and business logic boundaries, such as a central database service. Many microservice architectures depend on a single central database that stores the data that all other microservices process. When all this data is stored as a single data compartment, this would obviously be a great privacy hotspot. Applying micropseudonymization *within* this database service, however, could change this.

We could store the data for each microservice in different tables with different pseudonymization domains under different pseudonyms. The database itself does not need to be able to convert between the different pseudonymization domains, it just needs to store the data and perform simple queries. In this case, the database would have access to all data compartments and store their data, but is not able to link data between these compartments (only the services that depend on it). Notice that in this case, microservices are not able to exchange data with each other *via* this database or perform queries across compartments. A different middleware layer that integrates the pseudonymization service is required to facilitate this properly.

Micropseudonymization can thus be applied to separate data processing both *between (inter)* different microservices, and *within (intra)* a single microservice. By choosing pseudonymization domains smartly, micropseudonymization can be used to perform data minimalization, even within a single microservice.

4. Case study: research data platform

As presented in section 1, maintaining privacy in complex and evolving systems can be challenging. In this section, we present a specific use case where these challenges are especially relevant, and where micropseudonymization can be used.

Authors are involved in the development of a research data platform for the Dutch National Education Lab AI (NOLAI) at Radboud University that implements the PaaS architecture. Within NOLAI, many design-based research projects (10 new 3-year projects each year) are being conducted in cocreation between schools, researchers and businesses. Within these projects, prototype applications involving AI are being designed, developed, tested and scientifically validated within schools. These projects involve various forms of data processing in various different contexts.

For example, *operational* data for the primary functioning of the prototype is typically processed by businesses and schools. Meanwhile, parts of the data may be processed for *training* of AI models or general product development, by researchers and businesses. Finally, *scientific validation* data is collected to validate the educational effectiveness of the prototype. This may involve selected parts of the operational data, but may also involve additional data, such as survey data or background information about a student's learning deficiencies. Even more so, there may actually be several different research studies being conducted by different independent researchers, analyzing different subsets of the operational, training and scientific validation data.

In order to maintain privacy, it is crucial that these different forms of data are securely processed and properly governed. Survey data containing sensitive background information about a student's nationality, collected for scientific validation, for example, must be prevented from ending up in trained

AI models. Meanwhile, some data may *not* be shared with the researchers, while other data may *only* be shared with the researchers and not with the school. There is thus a complex system of authorized data flows between (sub)systems, hosted by different parties involved in these projects. Data is not processed in just a single research context, but in multiple different operational, product development and research contexts. Regular pseudonymization (replacing a direct identifier like a name for a single pseudonym that stays the same throughout all systems), as usual in scientific research, therefore does not suffice, as it does not prevent linkage of the different types of data between different systems.

System architecture The research data platform consists of several autonomous (sub)systems that can exchange data between each other. Examples include services for participant registration, surveys, file uploads, dashboards, long-term data storage, and various data analysis systems or integrations with prototype applications. They all have different business logic and trust boundaries between them, and are all involved in different research projects. With new projects starting every few months, the platform is always evolving and new systems are integrated frequently.

For each project, different pseudonymization domains are defined and associated with these 'users' (both functional systems, or individual researchers) that may get access to subsets of data from other 'users'. In principle, each service processes data with pseudonyms in its own domain. This does not need to effect the internal functioning of these services. For example, authors have integrated an existing LimeSurvey service as a data source via a API wrapper that only converts encrypted pseudonyms.

Only whenever data is exchanged between services, this goes through the central pseudonymization service (further just called PaaS). For example, when a research coordinator decides to start a survey, data from the participant registration service is sent to the survey system via PaaS and when a survey response is received for a specific participant, a notification is sent to the research progress dashboard, also via PaaS. And when a researcher then wants to analyze the survey responses together with log data uploaded to the file upload service, this also goes through PaaS. Each of these data flows must be explicitly authorized by PaaS, which only requires a simple configuration file to allow or disallow specific data flows. This centralizes data governance even though the systems themselves are decentralized.

Characteristics The PaaS architecture allows for rapid development, where different (micro)services can easily be developed and integrated while preserving privacy. We identify a few characteristics of this use case. We believe our insights hold generally for any systems where the architectural complexity and evolution of data processing may cause privacy challenges.

1. **High chance of compromise**: Because of the nature of cocreation, the variety and large number of projects, there is a wide variety of parties, systems and actors involved in the data processing. For each project, new data processing systems need to be set up (due to the nature of the projects) and the development speed of systems is high. Therefore, there is a high chance that at some point in time, one of these parties, systems or actors will be compromised (again, either intentionally or unintentionally) and their data is leaked. Since the *chance* is high, one can only maximally limit the *impact* of compromise to minimize privacy risk ($risk = chance \times impact$). Micropseudonymization does exactly this: it maximally limits the impact of data breaches without actually changing the attributes being processed in a specific microservice.
2. **Generic and practical solution**: Micropseudonymization is a generic privacy enhancing technology that can be implemented regardless of what form data processing actually takes within the individual microservices. It can be applied whenever a system has some form of functional compartmentalization and data exchange between compartments is properly implemented. This is important for this use case, because of the variety, scale and rapid development speed of systems. It is not feasible to spend significant time on implementing complex privacy-preserving computations that are specific to a single project. Micropseudonymization is a generic and practical measure, that is able to interface with both newly developed and existing systems.
3. **Centralized data governance**: Data governance suffers from a complex data processing architecture. When data is processed in numerous systems, with decentralized access control in each

of them, it becomes difficult to keep track of who may have had access to what data. Managing access rules and monitoring who accessed which data is challenging and error-prone when it is not centrally organized. By applying PaaS, access control on data linkability can be centralized despite the system architecture being highly decentralized.

A similar approach of pseudonymization using the PEP framework has been implemented in a long-term large-scale Parkinson's Disease Study [14], to disclose pseudonymous subsets of medical data to different research groups worldwide. Here, the system architecture itself is fixed and does not evolve, but the users and the datasets they request do evolve. In our research data platform, not only the users, but also the system architecture itself evolves. In both cases, central pseudonymization can be used to keep grip of data processing during evolution.

5. Future work

Data subject authentication So far, we discussed systems in which different microservices exchange data *about* data subjects with each other using pseudonyms, but we did not consider users communicating with microservice *themselves* (using pseudonyms). Whenever this needs to happen (e.g. our toy example in subsection 3.4), the user-interfacing service needs to authenticate the user. Extensions are required to enable the pseudonymization service to do this, similar to the goal of anonymous credential systems or typical pseudonym systems [15, 16, 17].

Pseudonymization service integrity While the PEP framework does realize privacy-friendly conversion of pseudonyms, it does not yet guarantee integrity or authenticity. Ongoing research by the authors, however, shows promising results using zero-knowledge proofs to enforce integrity throughout the pseudonymization process, where different distributed transcryptors can blindly verify each other that pseudonymization is performed correctly.

Distributed tracing Modern microservice architectures implement extensive monitoring tools to deal with the architectural complexity. An example is distributed tracing, where events or request are traced through different microservices, to see what goes wrong. These traces, though useful for debugging purposes, can be a big problem for privacy, as they enable data linkage and could even include user IDs directly. Pseudonymization of trace IDs, making them also domain-specific, may be an interesting approach for privacy-friendly distributed tracing, where trace IDs can only be linked through a pseudonymization service.

6. Conclusion

Privacy and security are cross-cutting concepts: they can not easily be added to a system but need to be implemented throughout the whole design. At its core, micropseudonymization solves the fundamental challenge that privacy becomes harder to maintain as system architectures become more complex. Traditional approaches to incorporate privacy may fail when system boundaries change. Micropseudonymization aligns privacy boundaries with security trust boundaries and business logic boundaries that microservice architectures already implement, implementing *privacy by design* by applying *pseudonymization by default*.

Within these compartments, micropseudonymization remains flexible and generic: the internals of a microservice are not adapted when micropseudonymization is implemented, except for some middleware to exchange data with other services. This makes micropseudonymization itself a generic and reusable component. It can be implemented in any well-designed data processing system that implements some form of functional compartmentalization, to limit the impact of data breaches. Moreover, by centralizing data governance, PaaS can be used to control and regulate data exchange to improve system monitoring, and to enforce important privacy features in complex and evolving data processing systems.

Acknowledgments

This research is performed in context of the Dutch National Education Lab AI (NOLAI), funded by the Dutch National Growth Fund.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] J. Doesburg, P. van Gastel, B. van Gastel, E. Poll, Data processing diagrams - A modeling technique for privacy in complex data processing systems, in: F. Bieker, S. D. Conca, J. M. del Álamo, Y. S. Martín (Eds.), Privacy and Identity Management. Generating Futures - 19th IFIP WG 9.6/11.7 and IFIP WG 11.6 International Summer School, Privacy and Identity 2024, Madrid, Spain, September 10-13, 2024, Revised Selected Papers, volume 705 of *IFIP Advances in Information and Communication Technology*, Springer, 2024, pp. 115–131. URL: https://doi.org/10.1007/978-3-031-91054-8_6. doi:10.1007/978-3-031-91054-8_6.

[2] C. Meijer, B. van Gastel, Self-encrypting deception: Weaknesses in the encryption of solid state drives, in: 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019, IEEE, 2019, pp. 72–87. URL: https://doi.org/10.1109/SP.2019.00088. doi:10.1109/SP.2019.00088.

[3] H. Nissenbaum, Privacy in Context Technology, Policy, and the Integrity of Social Life, Stanford University Press, 2009.

[4] D. J. Solove, A taxonomy of privacy, University of Pennsylvania Law Review 154 (2006) 477–564. doi:10.2307/40041279.

[5] L. Sweeney, Simple demographics often identify people uniquely, Carnegie Mellon University, Data Privacy (2000). URL: https://dataprivacylab.org/projects/identifiability/paper1.pdf.

[6] J.-H. Hoepman, Privacy design strategies, in: ICT Systems Security and Privacy Protection, IFIP Advances in Information and Communication Technology, Springer, 2014, pp. 446–459. doi:10.1007/978-3-642-55415-5_38.

[7] J. Camenisch, A. Lehmann, (Un)linkable pseudonyms for governmental databases, in: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, ACM, 2015, pp. 1467–1479. doi:10.1145/2810103.2813658.

[8] E. Verheul, B. Jacobs, Polymorphic encryption and pseudonymisation in identity management and medical research, Nieuw Archief voor Wiskunde 18 (2017) 168–172.

[9] European Data Protection Board, Guidelines 01/2025 on Pseudonymisation, 2025. URL: https://www.edpb.europa.eu/our-work-tools/documents/public-consultations/2025/guidelines-012025-pseudonymisation_en.

[10] European Network and Information Security Agency, Pseudonymisation techniques and best practices: recommendations on shaping technology according to data protection and privacy provisions, 2019. URL: https://data.europa.eu/doi/10.2824/247711.

[11] B. E. Van Gastel, B. Jacobs, J. Popma, Data protection using polymorphic pseudonymisation in a large-scale parkinson's disease study, Journal of Parkinson's Disease 11 (2021) S19–S25. doi:10.3233/JPD-202431.

[12] Logius - Ministerie van Binnenlandse Zaken en Koninkrijksrelaties, Privacy Impact Assessment DigiD Hoog, 2017. URL: https://www.eerstekamer.nl/overig/20190129/privacy_impact_assessments_pia.

[13] Logius - Ministerie van Binnenlandse Zaken en Koninkrijksrelaties, BSNk PP technische specificaties v11.1, 2024. URL: https://gitlab.com/logius/bsnk/bsnk-techspecs/bsnk/-/raw/main/BPTSlive.pdf.

[14] B. E. van Gastel, B. Jacobs, J. Popma, Data protection using polymorphic pseudonymisation in a large-scale parkinson's disease study, Journal of Parkinson's Disease 11 (2021) S19–S25. doi:10.3233/JPD-202431.

[15] D. Chaum, Security without identification: Transaction systems to make big brother obsolete, Communications of the ACM 28 (1985) 1030–1044. doi:10.1145/4372.4373.

[16] A. Lysyanskaya, R. L. Rivest, A. Sahai, S. Wolf, Pseudonym systems, in: Selected Areas in Cryptography, volume 1758 of *Lecture Notes in Computer Science*, Springer, 2000, pp. 184–199. doi:10.1007/3-540-46513-8_14.

[17] J. Camenisch, A. Lysyanskaya, An efficient system for non-transferable anonymous credentials with optional anonymity revocation, in: Advances in Cryptology — EUROCRYPT 2001, volume 2045 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 93–118. doi:10.1007/3-540-44987-6_7.

On the evolution of direct dependencies in npm packages

Shahin Ebrahimi-Kia[1], Jesus M. Gonzalez-Barahona[1], David Moreno-Lumbreras[1], Gregorio Robles[1] and Tom Mens[2]

[1] *Universidad Rey Juan Carlos, Madrid, Spain*
[2] *University of Mons, Belgium*

Abstract

Lehman's laws of software evolution postulate that software systems tend to grow in complexity and functional content. We aim to check if this growth can be observed when software applications are built as collections of reusable components, thus acknowledging that reused components contribute to growth and functional content. In particular, we study the evolution of the direct dependencies of JavaScript packages distributed through npm. We analyze this evolution with three different metrics capturing different growth patterns. Overall, we observe only small increases, suggesting that maintainers control the growth of the direct dependencies they rely on. We also find cases of increase/decrease patterns, which could signal specific efforts by package maintainers to reduce the number of dependencies.

Keywords

dependency analysis, software ecosystem, npm, Lehman's laws, software evolution, software complexity

1. Introduction

Package ecosystems of free, open source software (FOSS) components have revolutionized modern software development. Many software applications are no longer monolithic entities, but collections of software packages. Typically, the main application declares direct dependencies to packages to perform part of its functionality. In doing so, such applications have a new possible strategy for adapting to increasing complexity: instead of just incrementing their own code, they can rely on more packages [1]. In this work, we focus on the npm ecosystem of JavaScript/TypeScript packages because of its central role in web application development, and because it is the largest and fastest growing package ecosystem. For our analysis, we leverage on the fact that its public registry enables reproducible historical analysis.

Npm exhibits exponential growth [2], driven by the increasing reuse of packages to fulfill all kinds of functionality. This rapid growth resonates with Lehman's *laws of software evolution*, particularly those of *continuing growth*, *continuing change*, and *increasing complexity* [3]. These *laws* postulate that as software systems evolve, their complexity and interconnectedness naturally increases, demanding deliberate and continuous efforts to ensure stability and sustainability [4]. It remains an open question whether Lehman's insights also apply ecosystems like npm. Widespread availability of reusable packages makes the addition of direct dependencies a common maintenance action, which could make growth more likely, following *law of continuing growth*, which postulates that developers should constantly add new functionality to meet evolving requirements. But at the same time, the *law of increasing complexity* warns that uncontrolled dependency growth can lead to fragile applications, prone to cascading failures and difficult maintenance [3]. Ecosystems that fail to address these challenges risk accumulating technical debt, creating barriers to efficient software evolution [2].

In this paper, we study to what extent npm packages cope with complexity by adding new dependencies. We focus on *direct dependencies* because developers can control them directly: if

BENEVOL'25: Proceedings of the 24thBelgium − NetherlandsSoftwareEvolutionWorkshop, 17 − −18*November*2025, *Enschede*, *TheNetherlands*

shahin.ebrahimi@urjc.es (S. Ebrahimi-Kia); jesus.gonzalez.barahona@urjc.es (J. M. Gonzalez-Barahona); david.morenolu@urjc.es (D. Moreno-Lumbreras); gregorio.robles@urjc.es (G. Robles); tom.mens@umons.ac.be (T. Mens)

they want to "delegate" functionality to a third-party component, they just add it as a new direct dependency. While we acknowledge that *indirect dependencies* drive much of the ecosystem risk (such as security vulnerabilities and breaking changes), they are not under direct control of developers. We therefore state as our main research question RQ_1: **Do npm packages increase their number of direct dependencies over time?**

To answer RQ_1 we analyze 1,998 npm packages drawn from four public importance-oriented lists, for reasons explained in Section 3. For the selected packages, we study their number of direct dependencies over time. Since we are interested in characterizing evolution, we also need to define metrics to decide if packages are increasing or decreasing in direct dependencies, leading to the auxiliary research question RQ_0: **How can we characterise changes in the number of direct dependencies of packages as they evolve over time?**

2. Related Work

The complex dynamics of dependency networks, especially in the npm ecosystem, has been extensively studied. Decan et al. [2] highlighted that npm's granular packaging and reuse introduce fragility and increased vulnerability risks through deeper dependency chains. Zerouali et al. [5] identified technical lag, where many packages fall behind due to delayed updates. Moreno-Lumbreras et al. [6] employed visualization, including VR, to explore npm's networks and management shortcomings. Prana et al. [7] revealed that tools like Dependabot alert outdated dependencies but often miss systemic risks like cascading failures. Zahan et al. [8] and Cogo et al. [9] studied supply chain weaknesses and dependency downgrades, respectively. Decan et al. [2] also discussed semantic versioning usage, noting inconsistencies leading to conflicts, while flexible dependency declarations aid update adoption but affect stability.

Research on npm package selection for reuse includes Mujahid et al. [10], who identified documentation quality, GitHub stars, and security as key factors; Abdalkareem et al. [11] and Qiu et al. [12] examined trivial package use and popularity metrics. Chatzidimitriou et al. [13] used network analysis to identify clusters, while Haefliger et al. [14] studied reuse patterns.

Security concerns are paramount, with Vu et al. [15] studying dependency management risks. Wheeler [16] proposed diverse double compiling to detect compiler injections. Kabir et al. [17] found widespread neglect of good practices, with vulnerable dependencies being common. Scalco et al. [18] proposed LastJSMile to detect source and artifact discrepancies. Zerouali et al. [19] analyzed outdated packages and risks, underscoring the need for software integrity in evolving dependency networks.

Dependency evolution impacts software integrity and verifiability. Harrand et al. [20] emphasized monoculture dangers, showing widely used libraries create security bottlenecks and amplify vulnerabilities, advocating diversity for resilience. Goswami et al. [21] found 38% of npm package versions unreproducible due to flexible versioning and tool variation.

Insights from other ecosystems provide context. Raemaekers et al. [22] showed Maven updates often break compatibility; Bavota et al. [23] mapped interdependencies affecting Apache projects. Germán et al. [24] revealed distinct evolution patterns in the R ecosystem, with core packages stable and peripherals more churned.

To date, systematic validation of Lehman's laws [25] in npm is lacking. Our work empirically examines if npm's dependency growth fits Lehman's laws, especially *continuing growth* and *increasing complexity*. Wittern et al. [26] documented npm's growth and intensifying dependency interconnectivity from 2011 to 2015, aligning with Lehman's *increasing complexity*. Lehman's early work used releases and modules as proxies for time and size. Later studies [3, 27, 28] continued this, with some simulating calendar time [29, 30]. Studies by Godfrey and Tu [31, 32] and Robles et al. [33] focused on size and calendar time. Israeli and Feitelson [34] expanded metrics for Linux evolution, adding complexity and maintainability indices [35]. We extend software growth metrics by focusing on direct dependencies.

3. Data Preparation

To perform our analysis, we create a historical dataset of package releases with their relevant dependency growth metrics, following the steps presented in the following subsections. see details about the reproduction package at the end of this paper.

3.1. List of considered packages

Instead of considering any random package in the npm registry, we create a *purposive sample* by focusing on packages which are "relevant to production deployments". To this end, we use the lists in npm Rank[1]. Since these lists date from 2019, they exclude newer packages, or packages that were not relevant at that time, which is discussed in Section 7). However, this cut-off time also ensures that most packages will have several years of history.

npm Rank provides four lists of 1,000 packages each:

1. **Top 1,000 most depended-upon packages:** Packages with the highest number of other packages directly depending on them.
2. **Top 1,000 packages with the largest number of dependencies:** Packages with extensive dependency trees, relying on numerous other packages for functionality.
3. **Top 1,000 packages with the highest PageRank score:** A ranking that considers both direct and indirect dependencies, based on a network analysis metric similar to Google's PageRank. Packages with high PageRank scores often play a central role in dependency networks.
4. **Top 1,000 packages with the highest authority:** Based on the Hyperlink-Induced Topic Search (HITS) algorithm, which distinguishes between "hubs" and "authorities" in a network. Studying these authoritative packages helps identify core libraries that are central to the stability and resilience of npm's ecosystem [36, 37].

We combined all lists in one, removing all duplicates, resulting in a set of 2,480 unique npm packages. To retrieve the relevant metadata for each of these packages, we used Open Source Insights API[2]. This metadata includes a list of the identifiers for all its releases in npm, and the number of direct dependencies for each of those releases. For 30 packages we were not able to obtain the metadata (e.g., because the package is no longer available), resulting in a final dataset of 2,450 packages.

3.2. Data filtering

Given this list of packages, we want to have into account releases that are "the most suitable for use in production at any given time". We decided to only consider releases with a three-component *semantic versioning* (SemVer) identifier[3], since those are usually recommended for production. We thus excluded non SemVer-compliant release identifiers with custom versioning schemas or non-numeric suffixes, such as pre-release and post-release identifiers suffixes (`alpha`, `rc`, `pre`, `post` and so on). At any point in time, we consider only the highest published release (following SemVer order), to consider only the "front-wave" versions, excluding backported releases.

Figure 1 illustrates this filtering of release identifiers for two packages, Express and Webpack. We show three types of releases: SemVer in blue, non-SemVer in orange, and backports to lower branches in purple. Our filtering retains only SemVer releases on the highest major branch.

After this filtering, we also exclueded packages with less than six releases (452 packages), since those are too few to observe evolution. This resulted in a dataset of 1,998 packages and their SemVer-compliant release metadata and direct dependencies.

[1] https://gist.github.com/anvaka/8e8fa57c7ee1350e3491 Details and code: https://github.com/anvaka/npmrank
[2] https://docs.deps.dev/api/v3/
[3] https://semver.org/

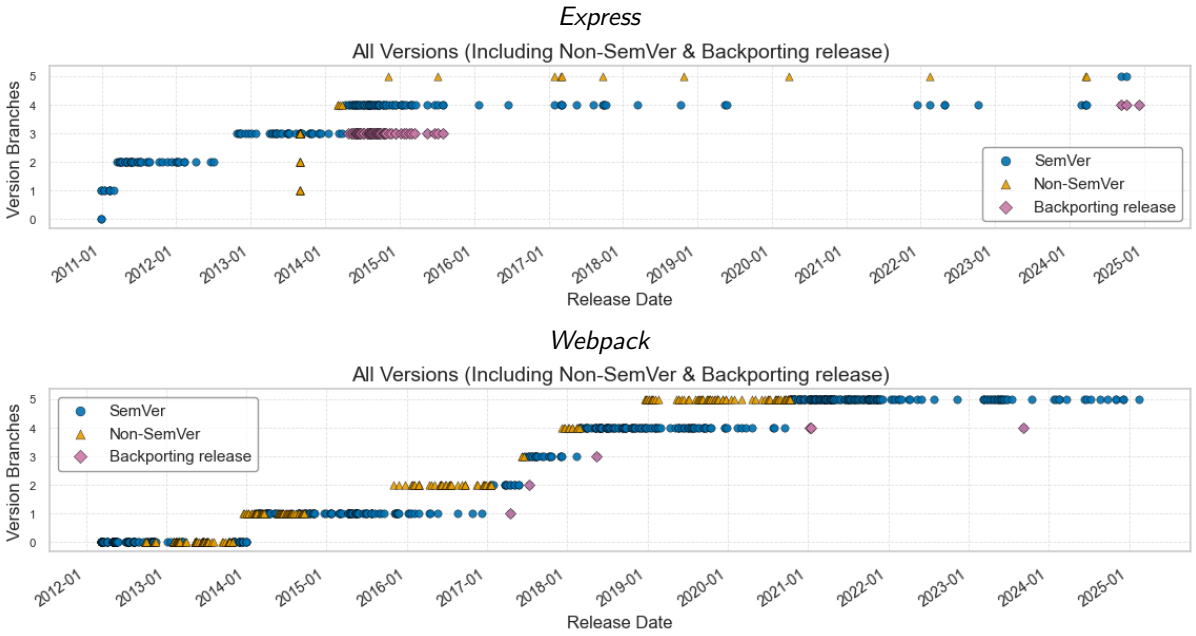
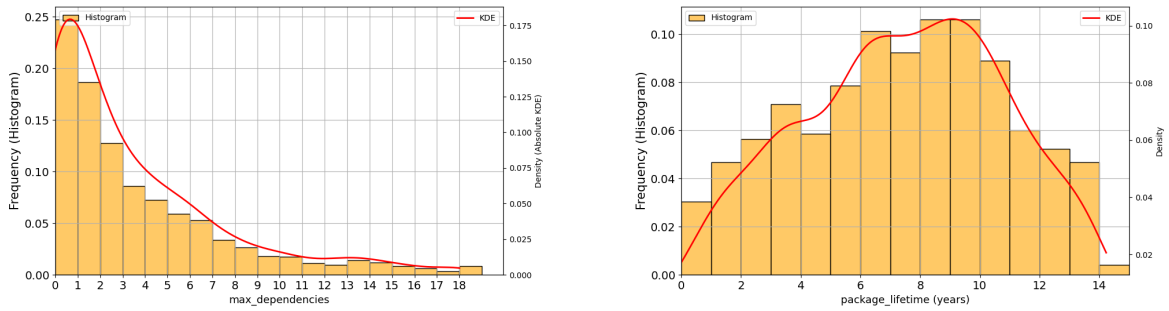


Figure 1: Identified release identifiers for `Express` (top) and `Webpack` (bottom). Each datapoint represents a release, with SemVer-compliant releases shown as blue circles, non-SemVer releases as orange triangles, and backporting releases as purple diamonds. This visualization clearly reveals the need to remove non-SemVer and backporting releases from the analysis.



(a) Max direct dependencies.

(b) Package lifetime (years).

Figure 2: Distributions for the considered package dataset (outliers excluded).

Figure 2a and Figure 2b show two distributions of aimed at characterizing and better understanding our dataset. After removing outliers, to allow for a better visualization, Figure 2a shows the distribution of the maximum number of direct dependencies for each package. While one out of four packages has one direct dependency, and the majority of packages has less than four direct dependencies, there is a long tail of few packages having a larger number of dependencies. Figure 2b shows the distribution of the package lifetimes (time period from the first to the last release for each package). The peak is around 9 to 10 years, although most packages have shorter lifetimes.

3.3. Metrics definitions

Understanding how the number of direct dependencies evolves as new releases of a package are produced is not simple. There are many situations to consider: for some packages the number of dependencies increases, and then remains stable, or maybe decreases afterward, or decreases and then increases again. For some packages, dependencies grow or shrink quickly, for some

others slowly. To take into account these nuances, we define multiple metrics, with the aim of characterizing the evolution of the growth of direct dependencies of a package release from different points of view.

To define these metrics, consider for each package release i a pair $(date_i, dep_i)$ in the Cartesian plane, where $date_i$ is the date of release i (represented in days since the date of the first release, i.e., $date_1 = 0$) and dep_i its number of direct dependencies. The pair corresponding to release 1 (the first one) is $(0, dep_1)$, and the pair for the last release n of the package is $(date_n, dep_n)$. Using this notation, we define the following metrics:

Absolute Change $= dep_n - dep_1$ Difference of direct dependencies between the last and the first release. This characterizes the net increase or decrease in number of dependencies.

First-Last Slope $= \frac{dep_n - dep_1}{date_n}$ Slope of the straight line from the pair corresponding to the first release, to the pair corresponding to the last release of the package. It measures the growth from the first release to the last one, taking into account the time span between them. Positive or negative values for this metric indicate an increase or decrease in dependencies. The larger numbers represent steeper increases or decreases.

Linear Slope $= \frac{\sum_{i=1}^{n}(date_i - \bar{date})(dep_i - \bar{dep})}{\sum_{i=1}^{n}(date_i - \bar{date})^2}$ ($\bar{x}$ stands for the mean of the distribution $x_1 \dots x_n$) Slope of the straight line resulting of performing a linear regression fit for the pairs corresponding to all releases of the package. This captures the overall trend, taking into account all releases.

Growth Fraction Fraction of the time during which a polynomial fit for all the pairs grows. We use a second-order polynomial regression, for characterizing the cases when there is a mix of increase / decrease periods with a single number which approximates the fraction of time dependencies are growing. The fitting is therefore a function of the form $y = ax^2 + bx + c$. Its derivative $y' = 2ax + b$ shows when the function increases or decreases, and when it changes from one to the other. This metric is computed as the period during which the derivative is positive, normalized between 0 and 1. 0 is for the case when there is no positive derivative, therefore the number of dependencies never grows, and 1 is for the case when the derivative is positive during the entire package lifetime, which means dependencies are always growing.

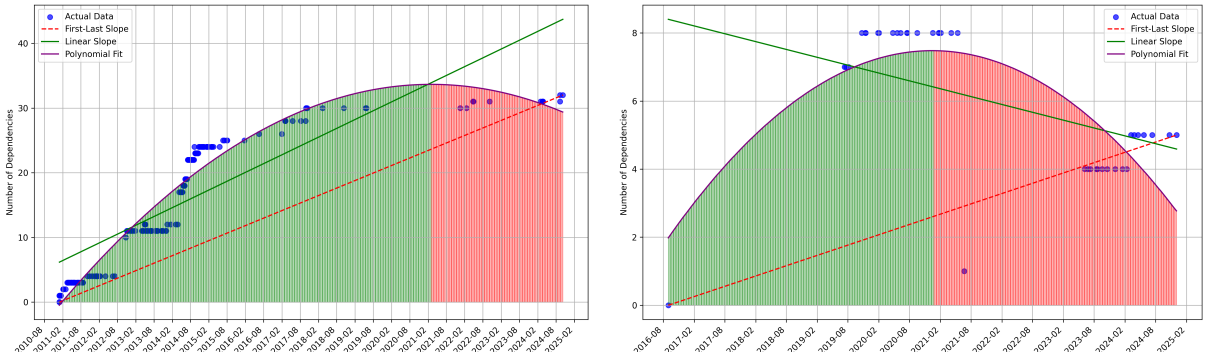
Package	Absolute Change	First-Last Slope	Linear Slope	Growth Fraction
`Express`	32	0.006	0.007	0.7350
`Web3-utils`	8	0.002	-0.001	0.5200

Table 1
Growth metrics for packages `Express` and `Web3-utils`.

Let us illustrate these metrics on our two example packages. All metrics values are reported in Table 1. Metrics values for other specific packages are available in the reproduction package.

Figure 3 shows that `Express` experiences a steady increase in direct dependencies, up to 2018, when it flattens. This is not captured by *Absolute Change*, *First-Last Slope*, or *Linear Slope*, which all just show increase. *Growth Fraction*, in this case, captures the two phases in the growth, but not exactly right: it signals correctly an increase up to 2021, but then it shows a decrease which is not really happening. However, it clearly shows that the trend is not just steady growth, as the other metrics show. `Web3-utils` starts increasing its dependencies (from 0 to 8) until later 2019, then remains stable, steps down sharply in mid 2021, and finally grows again. *Absolute Change* and *First-Last Slope* only increase, while *Linear Slope* shows decrease in dependencies. *Growth Fraction* captures the situation better, showing two phases, with a

decrease in the second which, even though not real, is still correct if we compare the plateau around 2020 to the final years.



(a) `Express` (b) `Web3-utils`

Figure 3: Visualisation of the elements used to calculate the four dependency growth metrics for two packages.

4. Analysis Results of the Dependency Growth Metrics

This section present the results of computing the growth metrics on the entire dataset. The descriptive statistics of all metrics are summarised in Table 2.

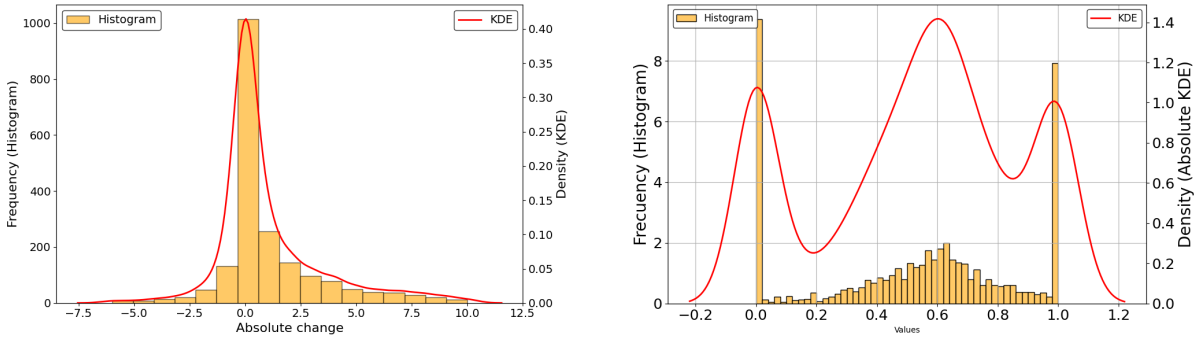
Table 2
Descriptive statistics for the four growth metrics for all considered packages

Metric	Mean	Median	Std. Dev.	Min	Max	Q1 (25%)	Q3 (75%)
Absolute Change	8.5143	0	27.9878	-310	462	0	4
First-Last Slope	0.15190	0	2.2360	-44.286	65.5	0	0.003
Linear Slope	0.16595	0	1.8994	-13.676	46.786	0	0.003
Growth Fraction	0.53288	0.5775	0.33468	0	1	0.315	0.775

To avoid distortion from extreme values, for all the charts below we applied interquartile range (IQR) filtering to remove outliers. This method removes values outside 1.5 times the middle 50% range, excluding extreme data while preserving typical behavior for accurate trends.

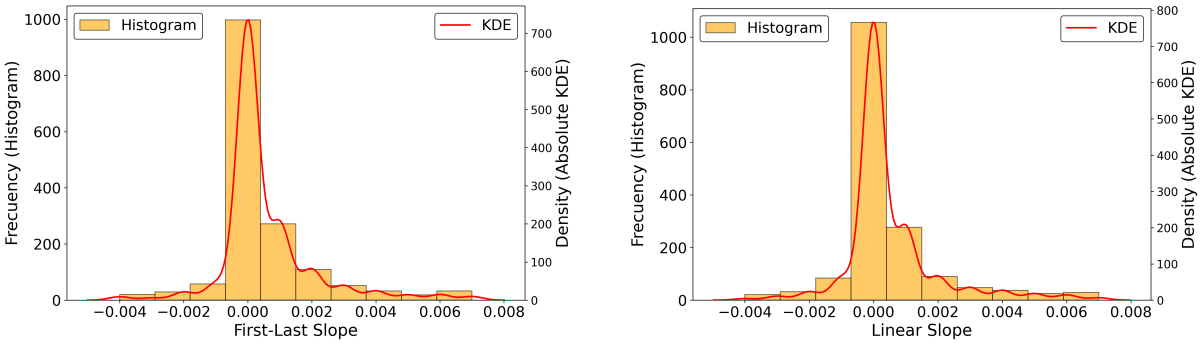
Absolute Change Figure 4(a) shows a histogram and a KDE of the distribution of *Absolute Change* values for all packages, after removing some outliers to make the visualization more clear. The distribution is centered around zero, with some skewness toward positive numbers (median is 0, as well as Q1, but mean is close to 8.5, and Q3 is 4). The most populated bucket is the one with a metric of 0 (more than 1,000 packages, or about half of all packages). The minimum and maximum observed values are -310 and 462. According to this metric, packages tend not to grow or shrink in number of direct dependencies, or do it very slightly. Most of them are quite grouped around a variation of 0 (the same number of dependencies at the start and at the end of the project). However, packages that grow tend to grow more than those that shrink. Some packages have huge variations in the number of dependencies, in the order of hundreds of them.

First-Last Slope Figure 5(a) shows a histogram and a KDE of the distribution of *First-Last Slope* values for all packages, after removing some outliers. The distribution is very much concentrated around zero (Q1 and median are 0, Q3 is almost 0 as well), very slightly skewed towards positive numbers. The central bucket, with slopes close to 0, is by far the larger. There are some outliers with extreme slopes (less than -44 and more than 65 for packages with



(a) Absolute Change (b) Growth Fraction

Figure 4: Histograms and KDEs of Absolute Change and Growth Fraction for all packages (excluding outliers).



(a) First-Last Slope. (b) Linear Slope.

Figure 5: Histograms and KDEs of First-Last Slope and Linear Slope for all packages (excluding outliers).

decreasing and increasing dependencies, respectively). This metric shows even more stability in the number of dependencies per package than the previous one. According to it, we could say that, by far, most packages remain stable over time, even when some of them grow or shrink, and only very few of them considerably. The growth from the first to the last release tends very strongly to zero.

Linear Slope Figure 5(b) shows a histogram and a KDE of the distribution of *Linear Slope* values for all packages, after outlier removal. The distribution is very much concentrated around zero, with Q1, Q3, and median 0 or almost 0, very slightly skewed towards positive numbers. The central bucket, with slopes close to 0, is by far the largest. There are some outliers with extreme slopes (less than -13 and more than 46 for packages with decreasing and increasing dependencies, respectively). There is a bit less dispersion than for *First-Last Slope* (standard deviation of 1.9 versus 2.2). Once again, this metric mainly shows that packages are stable in the number of their direct dependencies. There is even more concentration close to 0, with some extreme outliers. We can say that linear growth of direct dependencies tends very strongly to zero.

Growth Fraction Figure 4(b) shows a histogram and a KDE of the distribution of *Growth Fraction* values for all packages, after outlier removal. The distribution has two groups concentrated in the extremes (0 and 1), and then another one, much lower, point of concentration around 0.6. The distribution seems a bit skewed towards 1 (median is 0.58, mean is 0.53), and it is relatively symmetrical. According to this metric, packages seem to spend more time growing than shrinking in number of direct dependencies. However, there are more packages that reduce their dependencies than packages that increase them, if we consider only those who either reduce

or increase dependencies during all their life, respectively (values of the metric of 0 or 1).

5. Other observations

Our results show that, in general, packages tend to keep their number of direct dependencies stable. For cases with clear growth or decline, we investigated whether certain parameters influence the likelihood of increasing or decreasing dependencies. We focused on the *Linear Slope* metric to capture trends in dependency changes. We explored its relationship with maximum dependencies, package lifetime, and number of releases per package. Table 3 presents descriptive statistics for subsamples based on these criteria. Classification is based on median values: below median are "low", "short" or "few"; at or above median are "high", "long" or "many".

Table 3
Descriptive statistics of *Linear Slope* for specific populations of packages

Metric	Mean Slope	Median Slope	Std. Dev.
few Max Dependencies	0.0002	0	0.0009
many Max Dependencies	0.3386	0.003	2.7037
short Lifetime	0.3678	0.001	2.8225
long Lifetime	0.0015	0	0.0071
low Number of Releases	0.3418	0	2.8393
high Number of Releases	0.0263	0.001	0.1744

Packages with many dependencies show a higher mean slope (0.3386) than those with fewer (0.0002), indicating that more complex packages undergo more dependency changes. Similarly, packages with a short lifetime have a higher mean slope (0.3678) than long-lived packages (0.0015), suggesting that shorter-lived packages experience more dynamic dependency shifts. Finally, packages with a low number of releases exhibit a higher mean slope (0.3418) than those with a high number (0.0263), meaning that infrequently updated packages have more drastic variations. The median slopes remain close to zero across categories, showing that most packages maintain stable dependencies. In summary, we can state that complex, short-lived, and infrequently updated packages more often experience notable dependency shifts.

We also examined how the number of direct dependencies per package evolves over time. For this, we computed the mean and median number of direct dependencies for all packages at the end of every year. We considered, for each package, the latest available release at the end of the year (in other words, the latest release for that package that could be deployed at the end of each year). If the package didn't produce any release by that date, it is not considered. The results of this analysis are presented in Figure 6.

The figure shows a rapid increase in mean number of dependencies between 2015 and 2019, followed by a plateau starting around 2019. This plateau is likely to be influenced by the composition and timing of our dataset, which was produced according to the situation in 2019. Thus, packages that are "relevant" after 2019 are mist likely underrepresented, which could explain why the increasing trend does not continue after 2019.

Several phases can be identified in the evolution of the mean number of direct dependencies per package: *slow growth* until 2014, when both mean and median remain close to zero; *expansion* until 2019, with a sharp increase in mean direct dependencies, while median growth happens much more slowly; and *plateau* until the end of the time period, with both metrics remaining stable. In addition to the growth patterns, it is worth noticing that the increase in the difference between mean and median signals an increase in the spread of the distribution, with some packages evolving towards a very high number of dependencies.

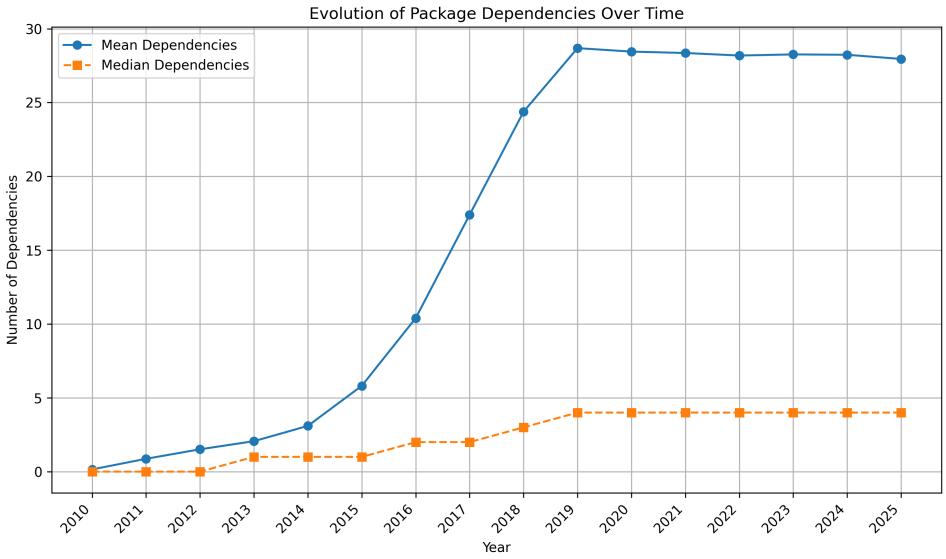


Figure 6: Mean and median number of direct dependencies over time.

6. Discussion

We answered RQ_0 by defining four metrics, designed to capture different aspects of the growth of direct dependencies, acknowledging the fact that there are many different growth patterns, and when one metric may identify growth, another one may identify decline. However, when we aggregate all packages, the results of these metrics mostly align. The two slope-based metrics have very similar descriptive statistics, and the other two show consistent distributions.

This allows us to answer RQ_1 in a conclusive way: the growth in number of direct dependencies over time in the considered packages is positive, but very close to zero. Therefore, Lehman's laws do not seem to apply to this kind of growth for our sample of packages. If JavaScript applications are growing, they will be in different ways (e.g. in terms of code size, or in terms of transitive dependencies, which have not been considered in this paper).

Of course, this doesn't mean that there are packages with steep increases or declines in the number of direct dependencies. We found some related parameters which could be a predictor for those cases, the most important of them being the age of the project (in number of releases or in time): young packages tend to grow, and grow faster.

From the point of view of developers, direct dependencies can be controlled directly by package maintainers. Therefore, their growth or decline falls completely under their responsibility. The results in this paper may help them to have a benchmark for comparison, so that they can analyze the evolution of their direct dependencies in the context of what is happening in comparable packages. In fact, this was one of the reasons why we included packages "relevant to production" in the dataset, instead of taking a random sample of packages: we want our study to be useful for practitioners, so we need to let them compare their components with relevant packages.

7. Threats to validity

Construct validity. We defined four metrics to quantify the temporal evolution of direct dependencies. Those metrics may not capture growth adequately, or may mask other effects which would lead to incorrect conclusions. However, we defined these metrics to capture different aspects of dependency growth, considering that the number of direct dependencies tends to be relatively low.

For measuring package growth we considered only releases with the highest SemVer identifier

when there was more than one development branch. This aims to capture the “front wave” of stable development, but may inaccurately reflect more complex development processes and usage patterns. It could happen, for example, that for long periods of time, new branches are ignored by users because they are still unstable, despite developers tagging them as stable releases. Future work could consider other ways of selecting the releases to measure growth.

We can also not discard errors in the data source we used for retrieving packages, and in the tools and scripts that we used for the analysis.

Internal validity. We selected the packages for building our base dataset based on several metrics of 'relevance'. But those metrics could not really represent relevance, or maybe relevance is not really relevant for this kind of study. Besides, the list is 5 years old, which means that the relevance of those packages may have shifted since then. Finally, the list is maybe too short fo be representative of 'relevant' packages. However, a manual inspection of it show many packages very relevant in the current software development, and the age of the list also allowed us to have packages with a certain lifetime, important to being able of measuring evolution in them.

We built our base dataset following a filtering process, trying to capture the real evolution of a package, removing all pre-release releases, and considering only versions "intended for deployment in production". We consider that our filtering strategy ensures that the dataset remains reliable and representative of real-world dependency trends, minimizing bias by eliminating unstable releases and outdated branches, But maybe this does not capture well the deployment practices of developers.

External validity. We cannot claim that all packages in any ecosystem behave the same way as the packages in our base dataset. In fact, we cannot even claim that in the case of npm, all packages behave like ours. We selected a very specific sample, trying to include in it packages relevant for production environments. From this point of view, even when we don't know if the patterns that we have found are common in other collection of packages, we think we can claim that we used a good sample of packages with an industrial interest, and therefore our results are likely useful for other similar packages, at least in the npm ecosystem. Of course, our analysis is limited in time. Even when the data is very recent (from March 2025), the list we are using is from 2019. It could happen that the evolution of the npm ecosystem is such, that our results are no longer valid for current 'relevant' packages.

8. Conclusion

To analyze how direct dependencies of npm packages evolve over time, we defined several metrics to consider different aspects of their growth. We measured them on a curated collection of npm packages, created with the main aim of including those relevant for production.

Understanding this evolution helps to understand to which extent Lehman's laws apply to applications built by composing packages, in which developers balance growth by adding more code of their own with growth by adding dependencies on reusable components.

Our findings reveal that, while most packages exhibit stability in dependency counts, a small subset shows either rapid growth or decline, maybe indicating shifts in software development practices. These variations highlight the diversity in software evolution, where some packages experience dependency expansion, while others undergo simplification or deprecation.

Acknowledgments. The research presented in this paper has been funded in part by the Spanish Ministerio de Ciencia e Innovación, through project Dependentium (PID2022-139551NB-I00). In addition, this research is supported by F.R.S.-FNRS research projects T.0149.22, F.4515.23 and J.0147.24.

Reproducibility and data availability. The data used in this paper, along with the final results, and the software written to produce these results, are available in a reproduction package[4].

[4] https://doi.org/10.5281/zenodo.15024304

References

[1] M. Vieira, D. Richardson, The role of dependencies in component-based systems evolution, in: International Workshop on Principles of Software Evolution, 2002, pp. 62–65.

[2] A. Decan, T. Mens, P. Grosjean, An empirical comparison of dependency network evolution in seven software packaging ecosystems, Empirical Software Engineering 24 (2018) 381–416. doi:10.1007/s10664-017-9589-y.

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

[4] I. Herraiz, D. Rodriguez, G. Robles, J. M. Gonzalez-Barahona, The evolution of the laws of software evolution: A discussion based on a systematic literature review, ACM Computing Surveys (CSUR) 46 (2013) 1–28.

[5] A. Zerouali, T. Mens, J. M. Gonzalez-Barahona, G. Robles, A formal framework for measuring technical lag in component repositories—and its application to npm, Journal of Software: Evolution and Process 31 (2018) e2157. doi:10.1002/smr.2157.

[6] D. Moreno-Lumbreras, J. M. González-Barahona, M. Lanza, Understanding the NPM dependencies ecosystem of a project using virtual reality, in: Working Conference on Software Visualization, 2023, pp. 84–92. doi:10.1109/VISSOFT60811.2023.00019.

[7] G. A. A. Prana, C. Bird, E. T. Barr, P. T. Devanbu, A. Hindle, Using practitioners' insights to investigate reproducibility of software engineering studies, Empirical Software Engineering 26 (2021) 1–37.

[8] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, L. Williams, What are weak links in the npm supply chain?, in: International Conference on Software Engineering, 2022, pp. 331–340.

[9] F. R. Cogo, G. A. Oliva, A. E. Hassan, An empirical study of dependency downgrades in the npm ecosystem, IEEE Transactions on Software Engineering 47 (2019) 2457–2470.

[10] S. Mujahid, R. Abdalkareem, E. Shihab, What are the characteristics of highly-selected packages? a case study on the npm ecosystem, Journal of Systems and Software 198 (2023) 111588.

[11] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, E. Shihab, Why do developers use trivial packages? an empirical case study on npm, in: Joint Meeting on Foundations of Software Engineering, 2017, pp. 385–395.

[12] S. Qiu, R. G. Kula, K. Inoue, Understanding popularity growth of packages in JavaScript package ecosystem, in: International Conference on Big Data, Cloud Computing, Data Science and Engineering (BCD), 2018, pp. 55–60. doi:10.1109/BCD2018.2018.00017.

[13] K. C. Chatzidimitriou, M. D. Papamichail, T. Diamantopoulos, N.-C. I. Oikonomou, A. L. Symeonidis, npm packages as ingredients: A recipe-based approach., in: ICSOFT, 2019, pp. 544–551.

[14] S. Haefliger, G. Von Krogh, S. Spaeth, Code reuse in open source software, Management Science 54 (2008) 180–193.

[15] I. Pashchenko, D.-L. Vu, F. Massacci, A qualitative study of dependency management and its security implications, in: ACM SIGSAC Conference on Computer and Communications Security, 2020, pp. 1513–1531.

[16] D. A. Wheeler, Countering trusting trust through diverse double-compiling, in: Annual Computer Security Applications Conference (ACSAC), 2005, pp. 13–48. doi:10.1109/CSAC.2005.17.

[17] M. M. A. Kabir, Y. Wang, D. Yao, N. Meng, How do developers follow security-relevant best practices when using NPM packages?, in: Secure Development Conference, 2022, pp. 77–83. doi:10.1109/SecDev53368.2022.00027.

[18] S. Scalco, R. Paramitha, D.-L. Vu, F. Massacci, On the feasibility of detecting injections in malicious npm packages, in: International Conference on Availability, Reliability and Security, 2022, pp. 1–8.

[19] A. Zerouali, V. Cosentino, T. Mens, G. Robles, J. M. Gonzalez-Barahona, On the impact of outdated and vulnerable JavaScript packages in Docker images, in: International Conference on Software Analysis, Evolution and Reengineering, 2019, pp. 619–623. doi:10.1109/SANER.2019.8667984.
[20] N. Harrand, Software Diversity for Third-Party Dependencies, Ph.D. thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2022.
[21] P. Goswami, S. Gupta, Z. Li, N. Meng, D. Yao, Investigating the reproducibility of NPM packages, in: International Conference on Software Maintenance and Evolution, 2020, pp. 677–681. doi:10.1109/ICSME46990.2020.00071.
[22] S. Raemaekers, A. Van Deursen, J. Visser, The Maven repository dataset of metrics, changes, and dependencies, in: Working Conference on Mining Software Repositories, IEEE, 2013, pp. 221–224.
[23] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, S. Panichella, The evolution of project inter-dependencies in a software ecosystem: The case of Apache, in: 2013 IEEE International Conference on Software Maintenance, IEEE, 2013, pp. 280–289.
[24] D. M. German, B. Adams, A. E. Hassan, The evolution of the R software ecosystem, in: European Conference on Software Maintenance and Reengineering, IEEE, 2013, pp. 243–252.
[25] M. M. Lehman, Programs, life cycles, and laws of software evolution, Proceedings of the IEEE 68 (1980) 1060–1076. doi:10.1109/PROC.1980.11805.
[26] E. Wittern, P. Suter, S. Rajagopalan, A look at the dynamics of the JavaScript package ecosystem, in: International Conference on Mining Software Repositories, ACM, 2016, pp. 351–361. doi:10.1145/2901739.2901743.
[27] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, W. M. Turski, Metrics and laws of software evolution—the nineties view, in: International Software Metrics Symposium, IEEE, 1997, pp. 20–32.
[28] M. M. Lehman, J. F. Ramil, An approach to a theory of software evolution, in: International Workshop on Principles of Software Evolution, 2001, pp. 70–74.
[29] B. W. Chatters, M. M. Lehman, J. F. Ramil, P. Wernick, Modelling a software evolution process: A long-term case study, Software Process: Improvement and Practice 5 (2000) 91–102.
[30] P. Wernick, M. M. Lehman, Software process white box modelling for FEAST/1, Journal of Systems and Software 46 (1999) 193–201.
[31] Q. Tu, M. Godfrey, Evolution in open source software: A case study, in: International Conference on Software Maintenance, IEEE, 2000, pp. 131–142.
[32] M. Godfrey, Q. Tu, Growth, evolution, and structural change in open source software, in: International Workshop on Principles of Software Evolution, 2001, pp. 103–106.
[33] G. Robles, J. J. Amor, J. M. Gonzalez-Barahona, I. Herraiz, Evolution and growth in large libre software projects, in: International Workshop on Principles of Software Evolution (IWPSE), IEEE, 2005, pp. 165–174.
[34] A. Israeli, D. G. Feitelson, The Linux kernel as a case study in software evolution, Journal of Systems and Software 83 (2010) 485–501.
[35] D. Coleman, D. Ash, B. Lowther, P. Oman, Using metrics to evaluate software system maintainability, Computer 27 (1994) 44–49.
[36] C. Ridings, M. Shishigin, PageRank Uncovered, Technical Report, Technical report, 2002.
[37] L. Page, S. Brin, R. Motwani, T. Winograd, The PageRank Citation Ranking: Bringing Order to the Web, Technical Report, Stanford Digital Libraries, SIDL-WP-1999-0120, 1999.

An Empirical Analysis of the GitHub Actions Language Usage and Evolution

Aref Talebzadeh Bardsiri[1], Alexandre Decan[1,2] and Tom Mens[1]

[1]Software Engineering Lab, University of Mons, Belgium
[2]F.R.S.-FNRS Research Associate

With the increasing demand for efficient and high-quality software systems, the practice of Continuous Integration, Deployment and Delivery (CI/CD) has become mainstream in software projects to streamline their development pipelines. CI/CD services automate repetitive tasks such as building code, running tests, and deploying applications. They have become an integral part of software development because they enhance productivity, improve efficiency, and reduce the likelihood of human errors [1].

In the past, different CI/CD services (e.g., Travis, CircleCI and Jenkins) were frequently used in GitHub repositories. Since the public release in 2019 of GitHub's own integrated CI/CD solution called GitHub Actions (hereafter shortened to GHA), it has become the most popular CI/CD tool on GitHub [2]. GHA allows repository maintainers to automate numerous activities, through YAML-based workflow configuration files.

For writing workflows, GHA provides a rich set of language constructs (i.e., keys, structures, values, etc.). According to Mernik's definition of a domain-specific language (DSL) [3], the GHA workflow syntax[1] is a DSL that allows workflow maintainers to define workflows, jobs, steps, and more. Its seamless integration with GitHub, its large marketplace of Actions, and its free plan for running workflows for public repositories, have made GHA a compelling choice among developers [2, 4].

However, beyond adoption, researchers have highlighted that its use comes with multiple challenges. Practitioners reported difficulties in understanding and writing workflow files. As an example, a workflow maintainer said *"YAML is untyped, which frequently leads to serious bugs in configuration code. I wish there was a statically-typed and more reliable alternative to YAML available and officially supported by GitHub Actions"* [5]. Ghaleb et al. [6] further found that GHA workflow files are among the most complex CI/CD automation services and can have high maintenance effort, while Zheng et al. [7] observed that GHA workflow files frequently fail during execution, highlighting challenges related to reliability and efficiency.

These findings suggest that GHA syntax and semantics may be poorly understood and insufficiently mastered by workflow maintainers. Despite the widespread adoption of GHA, there is a lack of comprehensive empirical studies that analyze the language constructs used in GHA workflow files, their usage patterns, and their evolution over time. We believe that addressing this gap is the first step for researchers to study current challenges in GHA workflows, such as their complexity and maintainability, and to provide a set of best practices to overcome these challenges.

In this empirical study of GHA language and its usage, we therefore aim to answer the following research questions:

RQ1 *What are the constructs of the GHA language?* A first step towards understanding the usage of GHA is to identify its language constructs. To this end, we conduct a large-scale empirical analysis of workflows and used the results as a proxy to enumerate the constructs of GHA. As an outcome of this RQ, we identify 197 constructs.

RQ2 *Which constructs are used in practice?* Understanding the usage frequency of the different GHA language constructs can reveal which constructs are central to workflow configurations and which

BENEVOL 2025: The 24th Belgium-Netherlands Software Evolution Workshop Enschede, 17-18 November 2023
aref.talebzadehbardsiri@umons.ac.be (A. Talebzadeh Bardsiri); alexandre.decan@umons.ac.be (A. Decan); tom.mens@umons.ac.be (T. Mens)
0009-0005-3719-9716 (A. Talebzadeh Bardsiri); 0000-0002-5824-5823 (A. Decan); 0000-0003-3636-5020 (T. Mens)

[1]https://docs.github.com/en/actions/reference/workflows-and-actions/workflow-syntax

ones are more specialized. To answer this question, we analyze the frequency of all constructs extracted from a large corpus of workflow snapshots. We conclude that only a small subset of them are frequently used in practice, while the majority occur rarely.

RQ3 *Which constructs contribute to GHA features?* To better understand the purpose of the different GHA constructs and to gain a higher-level view of the GHA language, we group constructs based on the features they contribute to. We refer to *features* as logical groupings of language constructs. For example, the *matrix strategy* is a feature that enables maintainers to run multiple instances of a job with different configurations and it involves several constructs. The outcome of this RQ is a mapping from constructs to features, along with an assessment of the number and nesting of constructs for each feature.

RQ4 *How are GHA features used in practice?* From RQ2, we observed that a small subset of GHA constructs are frequently used in practice. Building on this, understanding the frequency of usage of GHA *features* provides a higher-level perspective on how workflows are configured and implemented. We identify which features are commonly used and which ones are rarely observed in the workflows. In addition, we analyze to what extent workflows use the available constructs for each feature, and which constructs are most frequently employed in their implementation.

RQ5 *How does the GHA language usage evolve over time?* We analyze the evolution of GHA language constructs and workflow features to understand how its use changed over time. This knowledge can help us identify trends and change patterns in the usage of GHA.

To answer our research questions, we conducted an empirical study on a large-scale dataset of GHA workflow histories. The dataset, provided by Cardoen et al. [8], contains over 3 million workflow snapshots from 49K popular and active GitHub repositories related to software development, covering the period from July 2019 to August 2025. Such results pave the way for a more in-depth study on the complexity of writing and maintaining GHA workflow files.

Acknowledgments. This research is supported by F.R.S.-FNRS research projects T.0149.22 , F.4515.23 and J.0147.24.

References

[1] F. Zampetti, S. Geremia, G. Bavota, M. Di Penta, CI/CD pipelines evolution and restructuring: A qualitative and quantitative study, in: Int'l Conf. Software Maintenance and Evolution (ICSME), 2021.

[2] M. Golzadeh, A. Decan, T. Mens, On the rise and fall of CI services in GitHub, in: Int'l Conf. Software Analysis, Evolution and Reengineering (SANER), IEEE, 2022, pp. 662–672. doi:10.1109/SANER53432.2022.00084.

[3] M. Mernik, J. Heering, A. M. Sloane, When and how to develop domain-specific languages, ACM computing surveys (CSUR) 37 (2005) 316–344.

[4] A. Decan, T. Mens, P. Rostami Mazrae, M. Golzadeh, On the use of GitHub Actions in software development repositories, in: Int'l Conf. Software Maintenance and Evolution (ICSME), IEEE, 2022. doi:10.1109/ICSME55016.2022.00029.

[5] S. G. Saroar, M. Nayebi, Developers' perception of GitHub Actions: A survey analysis, in: Int'l Conf. Evaluation and Assessment in Software Engineering, 2023. doi:10.1145/3593434.3593475.

[6] T. Ghaleb, O. Abduljalil, S. Hassan, Ci/cd configuration practices in open-source android apps: An empirical study, ACM Trans. Softw. Eng. Methodol. (2025). URL: https://doi.org/10.1145/3736758. doi:10.1145/3736758, just Accepted.

[7] L. Zheng, S. Li, X. Huang, J. Huang, B. Lin, J. Chen, J. Xuan, Why do github actions workflows fail? an empirical study, ACM Trans. Softw. Eng. Methodol. (2025). doi:10.1145/3749371.

[8] G. Cardoen, T. Mens, A. Decan, A dataset of GitHub Actions workflow histories, in: Int'l Conf. Mining Software Repositories (MSR), ACM, 2024, pp. 677–681. doi:10.1145/3643991.3644867.

Language-Level Support for Multiple Versions for Software Evolution

Tomoyuki Aotani[1], Satsuki Kasuya[2], Lubis Luthfan Anshar, Hidehiko Masuhara[2] and Yudai Tanabe[2]

[1]Sanyo-Onoda City University, Yamaguchi, Japan
[2]Institute of Science Tokyo, Tokyo, Japan

Abstract

While versioned packages are widely used in today's software development, existing programming languages allow to use them on the 'one-version-at-a-time' principle, which makes software evolution inflexible. We proposed a notion called *programming with versions* that programming languages should allow to use multiple versions of a package and designed programming languages based on this notion. In this presentation, we overview the design principle of these languages and discuss how these languages can make software evolution more flexible and future challenges in programming language design.

Keywords

Programming with versions, versioned software packages, package management systems, VL, BatakJava, Vython

1. Introduction

A *versioned package* is a unit of managing programs for modular software development. From a programming language, a package is often associated with a module that provides a set of classes, functions, types, variables, etc. Outside of a programming language, a package is often managed by a *package manager* in which a *version number* of a package is used for distinguishing compatibility between different implementations of the same package. There have been many package managers developed for different programming languages such as npm[1], Gradle[1], PyPI[1], RubyGems[1], Cargo[1], to name a few.

Versioned packages play an important role in software evolution. Modules of a large-scale software system are often managed as versioned packages so that each package can independently evolve without concerning about the other packages. The notion of compatibility, i.e., whether one implementation of a package can be replaced with another implementation, makes the developers easier to expect if they can use a new implementation of a package without having serious problems. This is especially so when a software system uses third-party packages.

However, current programming languages are associated with versioned packages in a limited and less-flexible way because most languages can import merely *one version of a package at a time*. While this limitation is reasonable (if there had been two implementations of a function, which one would be used?), it causes many problems in software evolution, some of which can be exemplified by the following scenarios.

- Alice is developing a web application and wants to update its application framework to a new version for better rendering API. However, the new version has many incompatible changes and her team had to modify many places in the application code even though they are irrelevant to the new feature.
- Bob is developing an enterprise system that uses a version of a network library that known to have a security flaw. However, his team cannot switch the library to a new secure version for a long time because the new version has many incompatible changes.

BENEVOL 2025: The 24th Belgium-Netherlands Software Evolution Workshop, November 17–18, 2025, Enschede, The Netherlands
aotani@rs.socu.ac.jp (T. Aotani); satsuki.kasuya@prg.is.titech.ac.jp (S. Kasuya); masuhara@acm.org (H. Masuhara); yudaitnb@prg.is.titech.ac.jp (Y. Tanabe)

[1]https://www.npmjs.com/, https://gradle.org/, https://pypi.org/, https://rubygems.org/, https://lib.rs/crates/cargo

- Charlie is developing an arcade game and try to use a numerical package for improving character movements. After writing the simulation code, he finds that the graphics library used in the game requires an older version of the same numerical package. Unfortunately, those two versions are not compatible with each other, and the language requires to choose one of them.

2. Programming with Versions

The authors propose the notion of *programming with versions* (PwV), which is to support *multiple* versioned packages at the level of programming languages [1, 2, 3, 4]. The principle is to allow to use multiple versions of a package in one program. From the programming language design point of view, it is not difficult to design a language that can use multiple versions. It is more difficult to *prevent inconsistent usage of multiple versions.* For example, when we use two versions of an encryption package, a data encrypted by one version must be decrypted by the same version. To this end, we developed the following language designs.

- We designed a functional PwV language [1, 3] where every function can have different versions of implementations. Its type system guarantees that a program, even though it uses multiple versions of implementations, each data is processed by the functions that are implemented in the same version in order to maintain consistency.
- We designed a class-based PwV language based on Java [2] where not only functions but also data can have different version of implementations. Its type system guarantees that a bundle of data, i.e., an object, has its own version, and is always processed by functions, i.e., methods, implemented in the same version.
- We designed a dynamically-typed PwV language based on Python [4]. Rather than relying on type systems to guarantee version consistency, this language dynamically detects version inconsistency based on data provenance.

3. Software Evolution with Multiple Versions at a Time

Though PwV languages need more work on their design and implementation, those languages will bring more interesting challenges into software evolution. Given a greater freedom of using new versions of packages, we need to guide developers so that they can *gradually* incorporate new versions into their systems. Then notion of compatibility should also be reconsidered. For example, it is the developer who judge compatibility for the semantic versioning. When versioned packages are integrated into programming language semantics, we could mechanically judge semantic compatibility with a theoretical background.

As there have been proposed a tremendous amount of methodologies for software evolution (with traditional programming languages), revisiting those methodologies under the light of PwV languages would also be interesting.

References

[1] Y. Tanabe, L. L. Anshar, T. Aotani, H. Masuhara, A functional programming language with versions, The Art, Science, and Engineering of Programming 6 (2021).

[2] L. L. Anshar, Y. Tanabe, T. Aotani, H. Masuhara, BatakJava: an object-oriented programming language with versions, in Proceedings of the International Conference on Software Language Engineering, SLE 2022, 2022, pp. 222–234.

[3] Y. Tanabe, L. L. Anshar, T. Aotani, H. Masuhara, Compilation semantics for a programming language with versions, in Proceedings of Asian Symposium on Programming Languages and Systems (APLAS 2023), LNCS, 2023.

[4] S. Kasuya, Y. Tanabe, H. Masuhara, Dynamic version checking for gradual updating, Journal of Information Processing 33 (2025) 471–486.

On the Transferability of a Bot Detection Model from GitHub to GitLab

Cyril Moreau[1]

[1]*Software Engineering Lab, University of Mons, Belgium*

Abstract

Collaborative development platforms like GitHub and GitLab are central to software project lifecycles, but the increasing presence of automated accounts, or development bots, complicates the analysis of contributor behaviour. Existing development bot detection tools are primarily developed and evaluated on GitHub data, raising questions about their transferability to other platforms. This work investigates the transferability of BIMBAS, a state-of-the-art bot detection model for GitHub accounts, to GitLab. To make this transfer possible, we built the necessary tooling to extract and map GitLab user events into activity sequences, and constructed a ground-truth dataset of 593 annotated GitLab accounts (273 bots and 320 humans). Our experiments show that BIMBAS trained on GitHub achieves a weighted F1-score of 0.936 when applied to GitLab accounts. These results demonstrate that, although BIMBAS was designed for GitHub, it can be effectively transferred to GitLab, paving the way for more reliable empirical studies across platforms.

Keywords

GitHub, GitLab, bot identification, transferability, machine learning, automation

1. Context and findings

On collaborative development platforms such as GitHub and GitLab, developers heavily rely on automation mechanisms that support them in managing increasingly complex projects. To deal with repetitive, error-prone and time-consuming tasks such as testing or code reviewing, developers often use automated user accounts, commonly referred to as **development bots** [1]. The presence of these bots complicates empirical studies that rely on analysing human contributor activity, as the automated actions of bots can introduce significant biases [2]. Consequently, it is crucial to accurately identify automated accounts to distinguish human and automated activity.

Several tools for bot detection have been proposed in the literature, such as BoDeGHa [2], and more recently BotHunter [3] and RABBIT [1]. The latter leverages BIMBAS, a machine learning model based on user activity sequences generated through an activity mapping referred to as **rbmap** in this paper. However, these existing approaches have been designed and validated exclusively on public GitHub event data. To our knowledge, no automated bot detection tool has yet been systematically assessed on GitLab, raising the question of their generalisation to other collaborative development platforms, and highlighting the need for solutions dedicated to GitLab.

This work investigates the transferability of the BIMBAS model to GitLab. Transferring such a model is not straightforward, since differences in available events, recorded activities, and platform-specific practices require adapting the activity mapping and constructing an appropriate evaluation dataset. In this work, we therefore adapt the activity mapping to GitLab and build a ground-truth dataset of GitLab accounts, enabling us to evaluate whether a model trained on GitHub can effectively identify bots on GitLab.

Since the rbmap activity mapping does not accurately represent user behaviour and is difficult to adapt, Hourri et al. introduced **ghmap** [4], a more flexible mapping that better represents user behaviour. Building on this, we developed **glmap**, an extension of ghmap for GitLab, which enables BIMBAS to be applied on GitLab accounts. To evaluate the transferability of the model, we also built a new ground-truth dataset of 593 semi-automatically labelled GitLab accounts (273 bots and 320 humans).

BENEVOL 2025

*Corresponding author.

cyril.moreau@umons.ac.be (C. Moreau)

Table 1
Performances of BIMBAS with different training and test mappings on GitHub and GitLab datasets. Precision (P), Recall (R), and F1-score (F1) are reported for bots, humans, and a weighted average.

Mapping		Evaluated on	Bots			Humans			Weighted		
Train	Test		P	R	F1	P	R	F1	P	R	F1
rbmap	*rbmap*	*GitHub*	*.905*	*.891*	*.898*	*.896*	*.910*	*.903*	*.900*	*.900*	*.900*
rbmap	ghmap	GitHub	.888	**.883**	.885	**.887**	.892	.889	.887	.887	.887
ghmap	ghmap	GitHub	**.902**	.880	**.891**	.886	**.907**	**.897**	**.894**	**.894**	**.894**
rbmap	glmap	GitLab	.886	.980	.931	.983	.900	.940	.940	.935	.936
ghmap	glmap	GitLab	**.902**	**.984**	**.941**	**.986**	**.916**	**.950**	**.949**	**.946**	**.946**

As glmap relies on a different strategy than the original rbmap used in BIMBAS, we assessed the impact of this change by creating a variant of BIMBAS trained with ghmap, and comparing it with the original model. As shown in Table 1, performance differences are negligible. When BIMBAS is evaluated with ghmap but trained with rbmap, the weighted F1-score decreases by only 0.13. This gap is further reduced to 0.06 when BIMBAS is trained directly with ghmap. These results suggest that BIMBAS is insensitive to the choice of activity mapping, indicating that variations in the set of activities (such as those provided by GitLab) do not substantially affect its performance.

Concerning transferability to GitLab, one can observe from Table 1 that BIMBAS achieves a weighted F1-score of 0.936 when evaluated on GitLab accounts using glmap. When the model is trained with ghmap, which is conceptually closer to glmap, the score further improves to 0.946. These results indicate that a model trained exclusively on GitHub generalises well to GitLab, and that leveraging an activity mapping aligned with glmap further enhances cross-platform performance.

2. Conclusion

In this work, we introduced glmap, an activity mapping adapted to GitLab, and provided a ground-truth dataset of 593 GitLab accounts in order to evaluate the transferability of the BIMBAS bot detection model on GitLab. Building on these contributions, we showed that BIMBAS, originally designed for GitHub, maintains a weighted F1-score of 0.936, which demonstrates its transferability to GitLab. These findings highlight the feasibility of cross-platform bot detection and suggest many interesting directions for future work, most notably the construction of larger and more diverse datasets and the extension of the approach to additional collaborative development platforms.

References

[1] N. Chidambaram, T. Mens, A. Decan, Rabbit: A tool for identifying bot accounts based on their recent github event history, in: 2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR), 2024, pp. 687–691. doi:10.1145/3643991.3644877.

[2] M. Golzadeh, A. Decan, D. Legay, T. Mens, A ground-truth dataset and classification model for detecting bots in github issue and pr comments, Journal of Systems and Software 175 (2021) 110911. doi:10.1016/j.jss.2021.110911.

[3] A. Abdellatif, M. Wessel, I. Steinmacher, M. A. Gerosa, E. Shihab, Bothunter: An approach to detect software bots in github, in: 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), 2022, pp. 6–17. doi:10.1145/3524842.3527959.

[4] Y. Hourri, A. Decan, T. Mens, A dataset of contributor activities in the numfocus open-source community, in: 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR), IEEE, 2025, pp. 159–163. doi:10.1109/MSR66628.2025.00035.

Evolution-Resilient Class Contours

Mattia Giannaccari[1], Marco Raglianti[1]

[1]REVEAL @ Software Institute – USI, Lugano, Switzerland

Abstract

Analyzing large scale object-oriented software systems is complicated. Analyzing their evolution increases the complexity by one order of magnitude, due to the additional dimension of *time*. While software visualizations can help to analyze a single system snapshot, having informative evolution-resilient visualizations is challenging.

We present our recent work on Class Contours, a novel visualization metaphor that depicts source code entities as *building facades* by mapping domain properties as, for example, code-level features (*e.g.,* attributes, methods) on visual properties (*e.g.,* doors, windows). Architectural patterns (*as in urban architecture*) emerge naturally. We explore an evolution of the current implementation of Class Contours to include time in a flexible yet deterministic, informative, robust, and scalable way.

Keywords

Class Contours, Evolution-Resilient Software Visualization, Software Evolution

1. Introduction

Comprehending classes is critical to evolve a codebase [1]. When analyzing object-oriented software systems, developers need to reconstruct the role and behavior of a class in their mental models, from scattered fragments of code [2], visualized as multiple continuous pages of text. Their focus often wanders from packages in a top-down approach to classes in a bottom-up fashion, alternating between different knowledge retrieval strategies in an opportunistic way [3]. Gathering new knowledge about the system incrementally, by looking at overviews, is complemented by inspecting key points in detail for specific application logic hotspots that provide information about the system's inner working.

To address these needs we proposed Class Contours [4], leveraging the *building facades* metaphor to use simple 2D architectural elements, which represent features of the classes in an intuitive and coherent way. In the resulting architectural patterns (as in *urban architecture*), similar class roles correspond to similar buildings in the Class Contours overview. It becomes easier at this point to spot and analyze similarities and differences, to identify outliers and application logic hubs, to allocate attention to specific parts of the system according to the task at hand. Internal (*e.g.,* attributes, methods) and external structure (*e.g.,* clients, providers) appear on the facade of the building. For example, the number of lines of code is mapped to the width of the structure, attributes are represented as doors, and methods as windows, while, at a glance, the repeating glyphs start to form a pattern language.

2. Evolving ZION

We implemented the Class Contours metaphor [4] in a visualization tool, ZION (for which a tool demo is under review at ICSE 2026), to validate our approach. With respect to the previous publication we already improved its parsing mechanisms to more reliably extract class features (and provide better future cross-language compatibility) by substituting VerveineJ with CodeQL.[1] Meanwhile, we started to consider strategies to compare class contours of two versions of a system to highlight the evolution direction and which domain changes trigger a visually recognizable structural change. To achieve this goal for evolutionary analysis, two features are missing from ZION's current implementation.

BENEVOL'25: Belgium Netherlands Software Evoltuion Workshop, November 17–18, 2025, Enschede, The Netherlands

mattia.giannaccari@usi.ch (M. Giannaccari); marco.raglianti@usi.ch (M. Raglianti)

https://mattiagiannaccari.github.io (M. Giannaccari); https://www.inf.usi.ch/phd/raglianti/ (M. Raglianti)

0009-0008-9356-2921 (M. Giannaccari); 0000-0002-6878-5604 (M. Raglianti)

[1]VerveineJ: https://modularmoose.org/developers/parsers/verveinej/ – CodeQL: https://codeql.github.com/

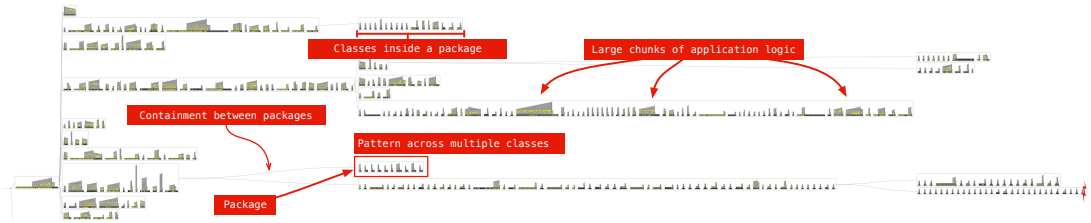


Figure 1: Hierarchical view of Class Contours in a tree layout of packages for `antlr4` classes.

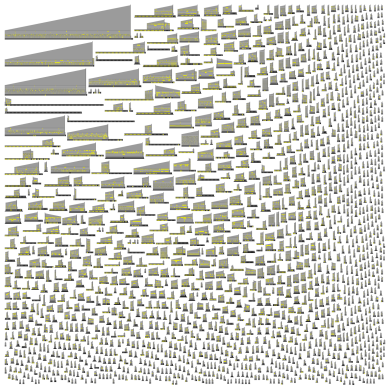


Figure 2: Class Contours rectangle packing (ArgoUML).

For positioning elements, ZION currently has just two layouts. A horizontal tree layout (Figure 1) to arrange Contours mirroring the structure of packages. And a rectangle packing layout (Figure 2) for space efficient placement in compact overviews, highlighting patterns at class level. We will discuss advanced potential layout strategies that remain consistent across different versions of the system, allowing to focus on the evolution of the contours themselves. While robust layouts that are evolution-resilient work for the city metaphor (*e.g.*, [5]), we would like to validate them in our simpler 2D representation.

The second missing feature is akin to normalization but involves deciding which metrics to visualize as *1-to-1 mapping* of the underlying feature and which are an *abstraction* of a core characteristic, independently from its "magnitude". The original intention of Class Contours was to represent the low level features in high fidelity when zooming in on a single building, while striking a convenient middle ground for scalability of overviews on large code bases. The new goal is to tackle the additional complexity of evolution, and time as a new dimension, while letting the Contours highlight important changes.

3. Conclusion

We present our current implementation of Class Contours, the recent update of the parser and its implications, while focusing on evolution-resilient layout strategies and normalization mechanisms to further extend the Class Contours beyond single snapshot analysis of a system. We sketch out for feedback the planned validation of our approach with the comparisons between UML class diagrams and class blueprints [6] on specific maintenance and evolution tasks.

Acknowledgments: This work is supported by the SNSF project "FORCE" (Project No. 232141).

Declaration on Generative AI: The author(s) have not employed any Generative AI tools.

References

[1] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Connallen, K. A. Houston, Object-Oriented Analysis and Design with Applications, 3rd ed., Addison Wesley, 2004.

[2] M.-A. Storey, Theories, methods and tools in program comprehension: Past, present and future, in: Proceedings of IWPC 2005, IEEE, 2005, pp. 181–191.

[3] M.-A. Storey, D. F. Fracchia, H. A. Müller, Cognitive design elements to support the construction of a mental model during software exploration, Journal of Systems and Software 44 (1999) 171–185.

[4] M. Giannaccari, M. Raglianti, M. Lanza, Skylines: Visualizing object-oriented software systems through Class Contours, in: Proceedings of VISSOFT 2025, IEEE, 2025, pp. 64–68.

[5] F. Pfahler, R. Minelli, C. Nagy, M. Lanza, Visualizing evolving software cities, in: Proceedings of VISSOFT 2020, IEEE, 2020, pp. 22–26.

[6] N. J. Agouf, S. Ducasse, A. Etien, M. Lanza, A new generation of CLASS BLUEPRINT, in: Proceedings of VISSOFT 2022, IEEE, 2022, pp. 29–39.

Preliminary survey on CPS testing in various domains of the industry

Guillaume Nguyen[1,*], Xavier Devroey[1]

[1]NADI, University of Namur, rue de Bruxelles 51, Namur, 5000, Belgium

Abstract

Cyber-Physical Systems (CPSs) help solve real-world challenges by gathering data and reacting physically to it in real-time. Through advanced driving assistance systems (ADAS), medical devices, or uncrewed aerial vehicles for agricultural purposes, CPSs are already well-present across various application domains. However, the testing techniques and strategies are often specific to those domains due to the versatile deployments of those systems. Furthermore, the constituent elements of CPSs are similar, so testing techniques from a specific domain could be adapted to fit the requirements of another one. In this paper, we perform a preliminary survey to probe the testing tendencies across CPS application domains.

Keywords

Survey, CPS, Testing, Standards, Regulations

1. Introduction

Cyber-Physical Systems (CPS) are ubiquitous and help solve daily challenges in many industry domains. From medical devices to advanced driving assistance systems (ADAS), they facilitate and enable the smooth operation of previously human-carried tasks. Of course, they should not endanger the security and safety of human bystanders (users, operators, patients, etc.). Rajkumar et al. define CPS as

> "*... physical and engineered systems whose operations are monitored, coordinated, controlled, and integrated by a computing and communication core.*" [1, p. 1]

Emphasizing applied CPS across different industries, we can see that the application domain is as wide as the number of human activities. Tekinerdogan et al. provide us with a general and complete feature model for CPS [2]. They also listed 10 application domains which seem relevant when looking at applications in the literature: **Health** where wireless medical devices presence is growing in hospital and operating rooms [3], **Smart Manufacturing** with smart factories through industry 4.0 which aim at increasing the efficiency of the product line either by reducing the costs or improving the flexibility of the resources by using interconnected devices, sensors and actuators [4], **Transportation** with advanced driver assistance systems (ADAS) alongside with other technologies aiming at self driving and connected cars [5], **Process Control** with the detection of chemical compounds in water for pollution detection and communication with waste-water plants [6], **Defence** with the upcoming of Lethal Autonomous Weapon Systems (LAWS) [7], **Building Automation** with connected devices to help monitoring and controlling a home and support the residents [8], **Robotic Services** in space exploration for example with the Ingenuity mars copter which had to adapt itself to the aerodynamics of Mars to perform the first successful flight there [9], **Critical Infrastructure** and the transitioning from classical grid management tools to smarter ones in order to improve the efficiency of a grid [10], **Emergency Response** by using CPS to increase the safety on construction sites [11], and **Other** for other types of CPSs.

Our research aims to find a test-oriented classification framework for CPS to perform efficient testing, considering the requirements and challenges of the various application domains. Indeed, as presented

BeNevol 2025: The 24th Belgium-Netherlands Software Evolution Workshop, November 17–-18, 2025, Enschede, The Netherlands
*Corresponding author.
guillaume.nguyen@unamur.be (G. Nguyen); xavier.devroey@unamur.be (X. Devroey)
0000-0002-9724-6634 (G. Nguyen); 0000-0002-0831-7606 (X. Devroey)

earlier, CPSs are classified mainly by application domain. However, the constituent elements for those systems are pretty similar (sensors, actuators, computing unit, etc.). Thus, the testing effort could be alleviated by consolidating the testing tools while keeping the domain-specific tests at later stages of the testing process. For this classification, we approached CPS from 3 central axes:

CPS Testing - How are CPS tested across industries?

CPS Engineering - How are CPS built across industries?

CPS Context - What are the non-functional requirements of CPS across industries?

2. Background

We understand that CPS are ubiquitous and benefit from a great versatility in their deployment. However, it seems to collide with other concepts from the industry, such as Embedded Systems, Internet of Things (IoT), real-time systems, or more generally operational technologies. Indeed, IoT is a "*concept used to define or reference systems that rely on autonomous communication of a group of physical objects*" [12]. We can see that IoT and CPS are related and should not be separated when defining and classifying CPSs. When bringing CPS and IoT together, Liu et al. suggest that CPS "*... deeply integrates the ability of computing, communication, and control based on information acquisition in IOT.*" [13, p. 28]. Lee and Seshia see CPS as an approach to embedded systems [14]. In their book, they define IoT as offering a means to interconnect sensors and actuators through networks to an interface inspired by the IT world, such as *Web Interfaces*. However, while IoT could fit the earlier definition of CPS, it is not suitable for time-critical real-world interactions. Indeed, *Real-Time Control* and *Safety-Critical Systems* require low-level logic and architectural designs. Lee and Seshia intend to give an introductory course on all the technical challenges of designing a CPS. In short, we could say that embedded systems are CPS components that use IoT to communicate.

The literature concerning testing and testing methods for CPS is quite complete. Indeed, Zhou et al. reviewed methods and test beds for testing CPS [15], they showed that CPS testing was a particularly rich field with many different and complementary techniques. Indeed, they list the following testing methods:

1. Model-Based testing (MBT), which is a formal method of checking the correctness of a model.
2. Search-Based testing uses meta-heuristics such as a genetic algorithm to generate test cases or test data automatically.
3. Online Monitoring, when complete formal verification is not possible, analyzing systems at runtime provides a formal technique that might leverage useful information.
4. Fault Injection testing, as producing failures artificially and consciously, speeds up the testing process.
5. Big Data Driven testing and prospects on using big data analysis by storing a large amount of data in CPSs.
6. Cloud Testing inspired by the advances of cloud computing and IoT.

They classify the testing methods into 4 objectives: **conformance testing**, **robustness testing**, **security testing**, and **fragility testing**. Of course, they also list the various contributions in terms of simulation-based testing, test-beds, and simulation-based test-beds for CPS by application domains, which have their own set of techniques and objectives.

3. Survey

The target population of the questionnaire was people working for Belgian or at least European companies in charge of governing whole or part of processes, including CPS design, development, test, and production, with a good knowledge of technical requirements, corporate internal processes,

industry standards, and legal requirements. The initial sampling intention is to target as many actors across industries as possible and provide an industry-specific overview of the more general challenges faced when dealing with CPSs.

The 53-question questionnaire was built using a French online tool called "Drag n Survey" that allows the user to drag fields onto a form and complete the questions. Due to license limitations, we could not use the automatic translation module, and we created three separate questionnaires in English, French, and Dutch.

The participants were solicited via a LinkedIn post, LinkedIn direct messages, contact lists from the Belgian CyberExcellence project, and contact lists from the computer science faculty of the University of Namur. We also participated in 4 industrial forums (2 local and 2 international) to interact with relevant companies directly.

Non-responses and dropouts were not monitored in real-time; however, as the level of knowledge required to answer the questions was relatively high, we assumed that participants might not have had sufficient knowledge to complete the questionnaire. As for missing data, we cleaned the data set of unusable responses.

We downloaded an Excel file with two sheets for all three questionnaires to analyze the responses. The "Questions" sheet has all the consolidated responses to the questions, with the number of responses to a specific question, for each multiple choice. The "Respondents" sheet or participants have all the individual responses to the questionnaire with IP addresses and time codes. A "+" sign separates multiple choices.

The extraction process is the following:

1. Load the three files in RStudio.
2. Load the questions from the "Respondents" sheet columns from the three questionnaires inside an R data frame.
3. In a new R data frame, load the information from the "Questions" sheet and assign question ID based on the "Respondents" data frame.
4. Extract and consolidate all the rows from the "Respondents" sheet from all questionnaires.
5. Remove NA rows and participants who said they couldn't answer the questionnaire from the final data frames.
6. Finally, identify dropouts and delete responses.

We had nine exploitable responses after soliciting respondents for 5 months and cleaning the data. It might be because it targeted C-level personnel with highly technical knowledge of the systems and corporate and regulatory expertise. They might not have the time to answer questions or be reluctant. However, we do not have enough information to elaborate more on that. On the other hand, 5 of those nine respondents agreed to be contacted for a case study, which is quite encouraging and will allow us to push the quality of our research further. We are well aware that this research constitutes preliminary research and should not be used to generalize the challenges regarding CPS across industries. However, it offers a nice probe for further study.

4. Results

In this section, we present the various results from our survey. The tables are gathered in Section A.

Out of the 9 respondents, we gathered responses from a telecom company marked as **Other**, a **Process Control**, a **Robotic Service**, three **Smart Manufacturing**, and three **Transportation** companies. We chose to present the following results by aggregating the data by domain of application. A description of the respondents is shown in Table 1. Even though we received 9 answers, we gathered a wide sample of industries and companies. All those companies used multiple devices at the same time, and sometimes more than 10,000 different devices. Only the robotic service and the telecom company didn't use interactive devices, while we suspect our question was not understood correctly. Although we formulated it as such: "*How many OT devices does your company use? For example, a smoke detection*

system with a smoke sensor, an alarm centre, and a sound alarm is composed of 3 devices." and "*As previously mentioned, those devices often interact with each other. Is that the case in your company?*" Systems used within those companies usually comprise devices from different manufacturers with proportions varying from 10 to more than 90%. Most of them also use industrial computers. Concerning the management and the number of departments using CPSs, we had different responses and mostly no answers.

Table 1
Overview of CPSs, interactions, and management by industry

Question	Other	Process Control	Robotic Service	Smart Man--ufacturing	Transporta--tion
Number of devices	1,000–10,000	>10,000	1,000–10,000	100; >10,000	10; >10,000
Device interaction	No	Yes	No	Yes	Yes
Avg. devices interacting	NA	10–100; 100–500	NA	10–100; 100–500	1–10; 10–100
Number of systems	NA	>100	NA	< 10; >100	< 10; >100
Systems with devices from different manufacturers	NA	50–90%	NA	10; >90%	10–50%
Industrial computer used	NA	Yes	NA	Yes	Yes
Same department manages systems	NA	Yes	NA	depends	No
Number of departments	NA	NA	NA	1; >3; NA	>3

Results for testing CPSs In Table 2, we can see that most respondents carry out tests at various levels, including functional and non-functional tests. Yet for the **Other** company, they don't seem to perform any testing themselves. In Table 3, we can see that most of them carry out tests before integrating a new device into their systems. They also perform quality insurance tests. Concerning the testing time spent at various phases of a product development, *design*, *development*, *prototype* and *production* we can see in Table 4 that **Smart Manufacturing** and **Transportation** perform testing from very short period of time to very long period of time at each phase while the **Process Control** company did not perform tests in production when the **Robotic service** company only carried out tests in production. In Table 5, we can see that various non-functional tests are performed at different phases of the development process.

Results for engineering CPSs In Table 6, we see that the **Smart Manufacturing** and **Transportation** companies have code developed internally, by manufacturers, and by third parties, which is consistent with previous answers. Concerning the Other company, the results are intriguing; they didn't seem to perform tests while they developed the code internally. Overall, the responses seem to be quite varied. Concerning programming languages and communication protocols presented in Table 7 and Table 8, the results are coherent with the industry standards, while we were surprised to find high-level languages such as Python, Java, and C# in the programming languages of CPSs.

Results for context surrounding CPSs The context in which the various companies operate follows the previous observations. Indeed, when looking at Table 9, the smart manufacturing and transportation companies with risk analyses at the various phases of the product development process follow multiple regulations and standards. However, every company only ticked the few regulations and standards we suggested, showing a misunderstanding of the regulatory and standardization landscapes of their industry. Free answers showed they had no idea about those, or they were following the provided requirements lists received from headquarters (in the case of international companies). When looking at the approval process in Table 10, we can see that there are multiple steps and multiple hierarchical levels involved with sensibly more complex processes for Transportation companies. This is consistent

with Table 11 showing a longer period of time required to introduce new devices or components within a system.

5. Conclusion

We presented results from 9 different companies grouped in 5 domains of industry. We encountered many difficulties in gathering information from multiple companies. We tried hard to interact directly with industrial actors during national and international forums, and we received enthusiastic responses from people met in person; however, we could never reach past the legal department of those companies. Furthermore, we never even reached a point where a non-disclosure agreement (NDA) was suggested.

Nevertheless, this survey offers interesting preliminary results showing the great variability in the companies using or developing CPSs. Interestingly, **Smart Manufacturing** and **Transportation** companies were particularly more tested, while we cannot reach conclusions with such a small dataset.

Concerning the future work, the *Context* surrounding the CPS development in the industry attracted our attention concerning the lack of understanding of the various regulations and standards applicable. This is similar to Zhou et al., who state that the CPS conformance (between a system and its specification) is not well exploited, probably due to the complexity of the various standards applicable to those systems [15]. Indeed, when looking at the regulatory landscape in the European Union, for example, multiple challenges arise [16]. Thus, we will investigate the CPS compliance verification capabilities in the industry. We will also contact the companies willing to perform a use case to continue gathering data on industrial CPSs.

Acknowledgments

This research was supported by the CyberExcellence by DigitalWallonia project (No. 2110186), funded by the Public Service of Wallonia (SPW Recherche).

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] R. Rajkumar, I. Lee, L. Sha, J. Stankovic, Cyber-physical systems, in: Proceedings of the 47th Design Automation Conference, ACM, New York, NY, USA, 2010.

[2] B. Tekinerdogan, D. Blouin, H. Vangheluwe, M. Goulão, P. Carreira, V. Amaral, Multi-Paradigm Modelling approaches for cyber-Physical Systems, Academic Press, San Diego, CA, 2020.

[3] M. R. Mahfouz, G. To, M. J. Kuhn, Smart instruments: Wireless technology invades the operating room, in: 2012 IEEE Topical Conference on Biomedical Wireless Technologies, Networks, and Sensing Systems (BioWireleSS), IEEE, 2012.

[4] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, M. Hoffmann, Industry 4.0, Bus. Inf. Syst. Eng. 6 (2014) 239–242.

[5] F. Arena, G. Pau, A. Severino, An overview on the current status and future perspectives of smart cars, Infrastructures 5 (2020) 53.

[6] V. Garrido-Momparler, M. Peris, Smart sensors in environmental/water quality monitoring using IoT and cloud services, Tren. Environ. Anal. Chem. 35 (2022) e00173.

[7] P. Scharre, Army of none: Autonomous Weapons and the future of war, W. W. Norton & Company, 2019.

[8] D. Marikyan, S. Papagiannidis, E. Alamanos, A systematic review of the smart home literature: A user perspective, Technol. Forecast. Soc. Change 138 (2019) 139–154.

[9] T. Tzanetos, M. Aung, J. Balaram, H. F. Grip, J. T. Karras, T. K. Canham, G. Kubiak, J. Anderson, G. Merewether, M. Starch, M. Pauken, S. Cappucci, M. Chase, M. Golombek, O. Toupet, M. C. Smart, S. Dawson, E. B. Ramirez, J. Lam, R. Stern, N. Chahat, J. Ravich, R. Hogg, B. Pipenberg, M. Keennon, K. H. Williford, Ingenuity mars helicopter: From technology demonstration to extraterrestrial scout, in: 2022 IEEE Aerospace Conference (AERO), IEEE, 2022.

[10] X. Fang, S. Misra, G. Xue, D. Yang, Smart grid — the new and improved power grid: A survey, IEEE Commun. Surv. Tutor. 14 (2012) 944–980.

[11] W. Jiang, L. Ding, C. Zhou, Cyber physical system for safety management in smart construction site, Eng. Constr. Archit. Manage. 28 (2020) 788–808.

[12] J. P. Espada, R. R. Yager, B. Guo, Internet of things: Smart things network and communication, Journal of Network and Computer Applications 42 (2014) 118–119. URL: https://doi.org/10.1016/j.jnca.2014.03.003. doi:10.1016/j.jnca.2014.03.003.

[13] Y. Liu, Y. Peng, B. Wang, S. Yao, Z. Liu, Review on cyber-physical systems, IEEE/CAA J. Autom. Sin. 4 (2017) 27–40.

[14] E. A. Lee, S. A. Seshia, Introduction to embedded systems, The MIT Press, 2.2 ed., MIT Press, London, England, 2017.

[15] X. Zhou, X. Gou, T. Huang, S. Yang, Review on testing of cyber physical systems: Methods and testbeds, IEEE Access 6 (2018) 52179–52194. URL: http://dx.doi.org/10.1109/ACCESS.2018.2869834. doi:10.1109/access.2018.2869834.

[16] G. Nguyen, M. Knockaert, M. Lognoul, X. Devroey, Towards comprehensive legislative requirements for cyber physical systems testing in the european union, 2024. doi:10.48550/ARXIV.2412.04132.

A. Tables

Table 2
Testing levels and types by industry

Industry	Testing levels	Functional tests	Non-functional tests
Other	NA	NA	NA
Process Control	Unit tests + Integration tests + System tests	Yes	No
Robotic Service	Integration tests + System tests	Yes	Yes
Smart Manufacturing	Unit tests + Integration tests + System tests	Yes	Yes
Transportation	Unit tests + Integration tests + System tests (sometimes only System tests)	Yes	Yes

Table 3
Integration tests and quality assurance practices

Industry	Integration tests before introduction	Quality assurance / testing on devices
Other	Yes	No
Process Control	Yes	Yes
Robotic Service	No	Yes
Smart Manufacturing	Yes	Yes/No (depending on case)
Transportation	Yes/No (depending on case)	Yes

Table 4
Available testing time per development phase

Industry	Design phase	Development phase	Prototyping phase	In production
Other	NA	NA	NA	NA
Process Control	1 week to 1 month	1 day to 1 week	1 week to 1 month	NA
Robotic Service	NA	NA	NA	Less than 1 hour
Smart Manufacturing	1 day to 1 year (depending on case)	1 day to 1 year (depending on case)	1 day to 1 year (depending on case)	1 day to 1 year (depending on case)
Transportation	1 day to 1 year (depending on case)	1 month to 1 year (depending on case)	1 month to 1 year (depending on case)	Less than 1 hour to 1 year (depending on case)

Table 5
Non-functional tests by industry

Industry	Non-functional tests performed	When performed
Other	NA	NA
Process Control	NA	NA
Robotic Service	System documentation compliance with actual behavior	Production
Smart Manufacturing	Load testing, Data security	Design + Development + Prototyping + Production
Transportation	Data security, System documentation compliance, Load testing	Prototyping + Production

Table 6
Summary of code development by industry

Industry	Internally	By manufacturers	By third parties
Other	X		
Process Control	X		X
Robotic Service		X	
Smart Manufacturing	X	X	X
Transportation	X	X	X

Table 7
Programming languages by industry

Industry	Programming languages used
Other	–
Process Control	ST, Ladder, FBD, C++, Python, Java, Javascript, C#
Robotic Service	–
Smart Manufacturing	C, Python, Java, Javascript, PowerShell/Script
Transportation	C, C++, Python, C#, ADA

Table 8
Communication protocols by industry

Industry	Communication protocols used
Other	–
Process Control	Modbus TCP, CAN, USB, Ethernet, MQTT, OPC UA, IP
Robotic Service	Ethernet, 4G/5G, Don't know
Smart Manufacturing	Modbus TCP, UART/USART, USB, Ethernet, MQTT, OPC UA, OPC DA, LoRA, 4G/5G, IP, Modbus Serial (disappearing)
Transportation	CAN, USB, Ethernet, MQTT, OPC UA, LoRA, 4G/5G, IP, MVB, CIP

Table 9
Risk analysis, regulations, and standards by industry

Industry	When do you perform a risk analysis?	Regulations / directives (laws)	Standards followed
Other	NA	NA	NA
Process Control	Design + Development	GDPR (EU 2016/679); NIS2 (EU 2022/2555); Regulation (EU) 2019/2144 (Automated driving system)	ISA/IEC 62443 (cybersecurity); ISO 27002 (information security management)
Robotic Service	Prototyping	NA	NA
Smart Manufacturing	Design + Development + Prototyping + Production	GDPR (EU 2016/679); NIS2 (EU 2022/2555)	ISA/IEC 62443; ISO 27002; Summary by HQ
Transportation	Development + Production; Prototyping + Production	NIS2 (EU 2022/2555); GDPR (EU 2016/679) + NIS2 (EU 2022/2555); IATF (PFMEA required)	ISA/IEC 62443; Internal; IATF

Table 10
Approval process: steps and people involved

Industry	Steps (approx.)	People (hierarchical levels)
Other	1	>3
Process Control	NA	NA
Robotic Service	2	2
Smart Manufacturing	1–3	2 to >3
Transportation	3–5	3 to >3

Table 11
Average time to introduce a new device or component

Industry	Average duration
Other	1 month to 1 year
Process Control	1 week to 1 month
Robotic Service	1 week to 1 month
Smart Manufacturing	1 day to 1 year (depending on case)
Transportation	1 week to >1 year (depending on case)

FlaDaGe: A Framework for Generation of Synthetic Data to Compare Flakiness Scores

Mert Ege Can[1], Joanna Kisaakye[1,2], Mutlu Beyazıt[1,2] and Serge Demeyer[1,2]

[1]Universiteit Antwerpen, Belgium
[2]Flanders Make vzw, Belgium

Abstract

Several industrial experience reports indicate that modern build pipelines suffer from flaky tests: tests with non-deterministic results which disrupt the CI workflow. One way to mitigate this problem is by introducing a flakiness score, a numerical value calculated from previous test runs indicating the non-deterministic behaviour of a given test case over time. Different flakiness scores have been proposed in the white and grey literature; each has been evaluated against datasets that are not publicly accessible. As such, it is impossible to compare the different flakiness scores and their behavior under different scenarios. To alleviate this problem, we propose a parameterized artificial dataset generation framework (FlaDaGe), which is tunable for different situations, and show how it can be used to compare the performance of two separate scoring formulae.

Keywords

Flakiness, Flakiness scores, Continuous Integration, Automation

1. Introduction

Software testing is a vital necessity for modern software engineering, ensuring system reliability, quality, and developer productivity in environments of increasing complexity [1, 2]. In continuous integration (CI) pipelines, automated test suites are executed to validate software works as intended before every deployment. However, these pipelines face a critical challenge: flaky tests [3, 4].

Flaky tests are tests with non-deterministic outcomes, switching between different results under identical conditions. This behavior reduces trust in the test results, wastes engineering efforts, and disrupts CI workflows [5, 6]. Large-scale industrial studies within firms such as Google, SAP and Microsoft show that flakiness continues to appear across projects and progressively worsens over time, causing significant costs in debugging and pipeline stability [7, 8].

Despite the increasing attention to flaky tests from the research community, there is still one principal challenge test engineers must grapple with, pertaining to the selection of a mitigation strategy. Before a mitigation strategy may be selected, the magnitude of the flakiness problem must be assessed. This is achieved through the use of flakiness scores and several scores exist in the white and grey literature [9, 10, 11, 12, 13, 14, 15]. However, most of the scores presented in the literature are evaluated against a dataset that is not accessible to the public. Indeed, at the time of this writing, there is no benchmark dataset against which multiple flakiness scoring techniques can be assessed.

This lack of standardized assessment frameworks allowing for a direct comparison of different flakiness scoring algorithms makes it harder to understand the strengths and weaknesses of the algorithms. Existing approaches are often evaluated in isolated configurations with unique use cases, making it difficult to establish a common baseline between algorithms. Moreover, a number of current research reports are based on datasets derived from real-world systems or undisclosed industrial case studies, which are individually valuable but do not always provide controlled conditions for systematic evalua-

The 24th Belgium-Netherlands Software Evolution Workshop (BENEVOL 2025)
m.egecan@gmail.com (Mert Ege Can); joanna.kisaakye@uantwerpen.be (Joanna Kisaakye); mutlu.beyazit@uantwerpen.be (Mutlu Beyazıt); serge.demeyer@uantwerpen.be (Serge Demeyer)
0009-0006-2595-0311 (Mert Ege Can); 0000-0001-7081-5385 (Joanna Kisaakye); 0000-0003-2714-8155 (Mutlu Beyazıt); 0000-0002-4463-2945 (Serge Demeyer)

tion. This gap hinders the ability to fairly assess the performance of competing algorithms in varying flakiness scenarios.

To address these challenges, this research introduces a configurable dataset generation framework (FlaDaGe) designed to simulate test suite behaviors with controllable flakiness trends. The framework is algorithm-neutral, which means that it is not biased towards any algorithm and allows for the systematic creation of artificial datasets with varying flakiness patterns. Finally, we demonstrate how it can be used to create a simulated dataset to compare two flakiness scoring algorithms; the No-Fault-Found (NFF) rate [11] and the Extended Flakiness Score (EFS) [12].

2. Flakiness Scores

Although re-running, monitoring, and fixing provide combined response strategies to flakiness, they all benefit from a unified quantification to rank which tests are most in need of attention. Flakiness scoring serves this purpose by calculating a numerical value for each test representing its level of instability [9, 10, 11, 12, 13]. The flakiness score of a test is derived from analyzing its execution history over a defined period, such as several days, weeks, or months in the CI pipeline. This score represents the degree of inconsistency in the results of a test, based on the ratio and frequency of change between the different test results. Tests that consistently produce the same result are considered stable, whereas alternating results are marked as more flaky.

In practice, a test with a high flakiness score means its results are unpredictable, and thus a candidate for re-run validation, long-term monitoring, or root cause analysis and fixing. For example, suppose that two different tests each fail three times in the last 20 runs. Test A fails in the first three consecutive runs, while test B fails sporadically such that failures occur in the third, eighth, and fourteenth runs. Although their failure counts are the same, Test B will exhibit more complexity, an unpredictable pattern when it fails, and therefore a higher flakiness score, reflecting its greater non-deterministic nature.

Flakiness scoring introduces several concrete benefits throughout the life cycle of software testing.

1. **Prioritization**: By being able to rank tests according to their instability, the development effort can be focused on the improvements with the greatest impact on the reliability of the system.

2. **Visualization**: Scoring enables graphical representations of flakiness statistics. Heat maps or historical flakiness plots offer valuable insight into the characteristics of flaky tests.

3. **Alerting**: Automated systems can flag tests whose flakiness score exceeds a predefined instability threshold, suggesting a review by the engineer.

4. **Tracking**: Flakiness scores can be tracked over time to assess performance in system improvements or component degradation.

Flakiness scoring also complements the re-run, monitor, and fix cycle. During re-runs, scoring helps decide whether additional executions are needed to reach a confident conclusion. In monitoring, scores support detecting trends depending on the change in score such as, constantly increasing values would mean instability is increasing as time advances. In the fix phase, the flakiness score can enable assessing the condition before and after the solution, validating whether the applied fix has effectively reduced the flakiness. Ultimately, flakiness scoring introduces an automated means of managing test instability in CI environments, allowing automated flaky test identification. In doing so, it strengthens the overall reliability and maintenance of the testing suite, ensuring that the effort spent is efficient and carefully maintained throughout the software delivery pipeline.

Table 1 shows a summary of the flakiness score definitions defined in the white and grey literature at the time of this writing. As shown in the table, the majority of flakiness scores proposed in the literature so far rely solely on the presence of test result history justifying the creation of an artificial dataset generation framework with which to generate data to compare different scoring algorithms.

Table 1
Comparison of flakiness score formulae and the data required to compute the score.

#	Paper (Year)	Formula	Required Data Elements
1	Kowalczyk et al.,(2020) [9]	Entropy: $f(R_{v,*}) = -\sum_{i\in(P,F)} p(i)\log_2 p(i)$ Flip rate: $f(R_{v,*}) = \frac{numFlips(R_{v,*})}{numPossibleFlips(R_{v,*})}$ Unweighted Average: $U_H(R) = \sum_{v\in V^t_{t-H}} \frac{f(R_{v,*})}{\lvert V^t_{t-H}\rvert}$ Weighted Average: $W_{\lambda,P}(R,n) = Z_n = \frac{\lambda x_n + (1-\lambda)Z_{n-1}}{\lambda\sum_{i=0}^{n-1}(1-\lambda)^i}$ where $x_n = \sum_{v\in V^{nP}_{(n-1)P}} f(R_{v,*})$	Test Result History. Weights.
2	Gruber et al.,(2023) [10]	Flip rate: $flip\_rate(R) =$ $\sum_{t=1}^{n-1}\left(\frac{1}{n-1}\cdot\begin{cases}1, & \text{if } r_t \neq r_{t+1}\\ 0, & \text{if } r_t = r_{t+1}\end{cases}\right)$ Decayed Flip rate: $flip\_rate(R,w) =$ $\sum_{t=1}^{n-1}\left(\frac{w(t)}{\sum_{u=1}^{n-1} w(u)}\cdot\begin{cases}1, & \text{if } r_t \neq r_{t+1}\\ 0, & \text{if } r_t = r_{t+1}\end{cases}\right)$	Test Result History. Weight functions and Weights.
3	Rehman et al.,(2021) [11]	NFF rate: $\text{NFFRate}_t = \frac{f}{r}$ Stable NFF rate: $\text{StableNFFRate}_t = \text{NFFRate}_t(f,r)$ Likelihood: $P_t(f,r,p) = \binom{r}{f}\cdot p^f\cdot(1-p)^{r-f}$ where $p = \text{StableNFFRate}_t$	Test Result History. Test Report History (Whether an issue or bug was filed for the test).
4	Kisaakye et al., (2024) [12]	Transition rate: $T(R_{v,*,\{r_1,r_2\}}) = \frac{numTransitions(R_{v,*,\{r_1,r_2\}})}{numTotalTransitions(R_{v,*})}$ Multi transition rate: $\sum_{\{i,j\}\subseteq\{r_1\ldots r_n\},i\neq j} T(R_{v,*,\{i,j\}})$ Unweighted Average: $U_H(R) = \sum_{v\in V^t_{t-H}} \frac{f(R_{v,*})}{\lvert V^t_{t-H}\rvert}$ Weighted Average: $W_{\lambda,P}(R,n) = Z_n = \frac{Y_n}{\lambda\sum_{i=0}^{n-1}(1-\lambda)^i}$ where $x_n = \sum_{v\in V^{nP}_{(n-1)P}} \frac{f(R_{v,*})}{\lvert V^{nP}_{(n-1)P}\rvert}$ and $Y_n = \lambda x_n + (1-\lambda)Y_{n-1}$	Test Result History. Weights.
5	Facebook/Meta Eng. Blog, (2020) [13]	Probabilistic Flakiness Score: $\text{PFS} = P(\text{Failure} \mid \text{good state})$ (estimated via Bayesian model).	Test Result History. context/features to distinguish "good" vs "bad" state Bayesian priors for model parameters.
6	Rasheed et al., (2020) [14]	Flakiness Score: The variability between test runs.	Test Result History.
7	Haben et al., (2024) [15]	Flake rate: $flakeRate(t,n) =$ $\frac{1}{w}\sum_{x=n-w}^{n-1} flake(t,x)$ Where $flake(t,x)$ is defined as: $flake(t,x) =$ $\begin{cases}1, & \text{if test } t \text{ flaked in build } b_x\\ 0, & \text{otherwise}\end{cases}$	Test Result History.

3. Constructing a Fair Evaluation Ground

This study has one driving research question.

How can we design a unified and statistically controlled dataset that enables a fair and algorithm-neutral comparison of different flakiness scoring algorithms?

To compare two fundamentally different scoring models, a shared evaluation ground is required. This dataset should not encode the assumptions or internal mechanics of any algorithm. In response to this, we propose a framework to enable the generation of an artificial dataset designed to simulate flakiness patterns in a controlled and observable way. Test executions are modeled across multiple versions, and result sequences are generated using probabilistic functions that reflect common flakiness patterns.

The dataset should also support the input requirements of all currently available algorithms. By examining Table 1, we can see that most flakiness algorithms require a test result history and one a test report history. For this purpose, we select one algorithm as a representative for all algorithms that require a test result history, the one proposed by Kisaakye et al. [12], and another representative of those algorithms that require test report presence, Rehman et al. [16]. By studying the requirements of these two algorithms, we are able to gather the broad spectrum of requirements for an artificial dataset generation framework.

4. Requirements for a Dataset

Before compiling the requirements for a dataset generation framework, it is essential to understand the characteristics of the datasets originally used in the evaluation of the representative flakiness scoring algorithms.

- Rehman et al. evaluated the NFF algorithm using industrial-scale test execution data collected from Ericsson's CI environment [16]. Crucially, each test run is labeled with one of the two possible binary outcomes, Pass and Fail, and information about the presence of fault reports, aligning with the definition of flakiness used by the NFF algorithm, failures without a report.
- Kisaakye et al. evaluated the EFS algorithm against artificial datasets designed to emulate flakiness using controlled statistical parameters [12]. The first is created using the dataset generation algorithm proposed by Kowalczyk et al., and models flakiness using probability mass functions in which case each test case has a static flakiness probability, and each version has a probability of revealing a fault [9]. The second adds support for more test results, Error and Skip, and more complex flakiness trends such as Increasing, Decreasing, Sporadic, Sudden spikes or drops.

Therefore, to allow a valid comparison between such models, a "good" dataset must meet these essential criteria:

- **Different Result States:** The dataset should include different result states to model real world build systems that have more results than Pass and Fail.
- **Report Association:** Some failure states must include report flags, and others must lack them, to simulate realistic distributions of fault attribution. This is a necessary requirement for all flakiness scoring algorithms that define flakiness as a failure in the absence of a test report.
- **Varied Flakiness Trends:** Flakiness should evolve over time using various trend profiles, such as increasing, decreasing, or stable, to test the ability of each algorithm to react to environmental variations.
- **Version and Run Structure:** The dataset must have a clear definition of version and run to allow scoring models to assess the historical progression of the test behavior. Each test should have multiple runs per version, for multiple versions.

5. Artificial Dataset Generation Framework

The dataset generation framework is built around six concepts: *Test*, *Suite*, *Version*, *Run*, *Trends*, and *Report Presence*. These concepts are often represented intuitively in the literature, however, they require clear definitions since they are necessary for most of the flakiness scores presented in Table 1. They aid in modeling different dimensions of flakiness behavior, and, when systematically combined, simulate evolving test flakiness.

5.1. Test

A *Test* represents the fundamental unit of analysis in the dataset. It encapsulates flakiness flags, flakiness probability, flakiness probability change (δ), and a population of versions.

Within this framework, each test is represented as a simulated execution trace spanning multiple versions each with hundreds of individual runs. Each test is assigned a fixed flakiness flag at the beginning of the dataset generation process: *clear*, *faulty* or none which means the test is *flaky*. *flaky* tests are eligible to receive flaky results, while *clear* and *faulty* tests are constrained to emit only Pass or Fail results, respectively. The probability of flakiness for *flaky* tests is randomly assigned based on predefined thresholds. This flakiness probability is forwarded to versions where the effect of the version trend is calculated, and determines the occurrence of a flaky result for every run generated within the version. The distinction between flaky and non-flaky tests ensures that the ground truth flakiness of each test is known in advance, forming the basis for evaluating how accurately an algorithm can differentiate this status based solely on test results.

5.2. Suite

A *Suite* is the highest-level organizational unit in the dataset generation framework. It encapsulates an entire simulation scenario defined by a unique combination of flakiness trends and execution parameters. Each suite includes a population of tests and a consistent configuration of version and run trends, thereby modeling a testing environment.

5.3. Version

A *version* in the framework represents a high level temporal segment within the dataset, simulating a specific period in the development cycle. The *version* is equivalent to a unique state of the software product within a CI system, capturing a specific execution configuration of the whole ecosystem, and acting as a reproducible environment for automated test execution. It can be represented by various identifiers depending on the development or deployment environment used, such as a build number or a tagged release.

The first version typically serves as a control baseline, while subsequent versions re-calculate the version-level flakiness probability according to version trend. The change on top of the test-level flakiness probability depends on the selected version trend, and is calculated using the δ value and the ratio of current version to total version count. By modeling multiple versions, the dataset generated reflects the evolving nature of development workflows.

5.4. Run

A *run* within the framework is the smallest unit of the dataset that carries the core information of each individual test execution performed for a test in each version. This concept corresponds to a run within a CI system which is a single execution of a test within a particular version. When a *run* is generated, the result is decided independently according to a pre-defined flakiness probability assigned to the test for the current version.

Run-level flakiness probability, is calculated in a manner similar to version-level flakiness probabilities according to the selected run trend of the suite. At this level, the run scales the version-level flakiness probability using the ratio of the current run to the total runs.

5.5. Trends

Once the test results are organized by version and run, it is possible to interpret and characterize emergent flakiness behaviors as *trends*. Trends provide time-wise information about test stability. Recognizing these trends allows test engineers to correlate the evolution of flakiness and take smarter precautions or apply more targeted solutions.

Flakiness trends can be analyzed at two levels; across *runs*, i.e, **run trend**, how individual test results evolve over multiple executions within the same version, and across *versions*, i.e., **version trend**, how the overall flakiness score for a given test changes from one version to the next. Table 2 summarises the trends implemented within the framework and how they affect flakiness behavior at the version and run level. The trends presented in Table 2 build upon those presented in [12] and aim to generalise the flakiness patterns found in practice. For example, the application of a direct fix for a single test within a single version, such as a code change, should trigger a sudden drop in test flakiness, while the application of an indirect fix, such as network stabilization, may only be observed as gradually decreasing flakiness.

5.5.1. Version Trend

The trend of the version defines the evolution of the flakiness by specifying a δ value and changing the base flakiness of each test by that amount accordingly.

These patterns are configured globally per suite and applied across all flaky tests in that suite. This ensures statistical consistency while maintaining a controlled environment for evaluating the performance of the algorithms. The dataset generation process implements the trends in Table 2 as three types of change patterns: *linear*, *exponential*, and *sudden*, which control how the δ value is applied to intermediate versions:

$$\text{Linear:} \quad p_v = p_t \pm \delta \cdot \frac{v}{V} \tag{1}$$

$$\text{Exponential:} \quad p_v = p_t \pm \delta \cdot \left(\frac{v}{V}\right)^2 \tag{2}$$

$$\text{Sudden:} \quad p_v = \begin{cases} p_t, & v < T \\ p_t \pm \delta, & v \geq T \end{cases} \tag{3}$$

where p_v is the version-level flakiness probability v, p_t is the test-level flakiness probability, V and v the total number and the current number of versions, respectively. T is the threshold version and, when reached, the δ value is applied in full. The positive and negative signs in the formulae depend on whether the trend is *increasing* (positive) or *decreasing* (negative).

As an example, if a test is assigned a test-level flakiness probability of 0.3 and the δ value is set to 0.15, then:

- For a decreasing trend, the probability that the final version is reached would be $0.3 - 0.15 = 0.15$.
- For an increasing trend, the probability that the final version is reached would be $0.3 + 0.15 = 0.45$.
- For a uniform trend, the probability would remain constant at 0.3.

Table 2
Trends implemented within the framework and the induced behavior at the Version and Run level

Trend	Version	Run
Uniform	Every flaky test retains its initial flakiness probability.	The flakiness probability does not change within a version. Flaky instances are spread randomly across the run sequence.
Increasing	The initial flakiness probability of every flaky test increases linearly across versions.	The flakiness probability starts from the probability 0 and linearly increases to the version level flakiness probability. Flakiness is concentrated in the later runs.
Decreasing	The initial flakiness probability of every flaky test decreases linearly across versions.	The flakiness probability starts from the version-level flakiness probability and linearly decreases down to 0 probability. Flakiness is concentrated in earlier runs.
Exponentially Increasing	The initial flakiness probability of every flaky test increases exponentially across versions.	The flakiness probability starts from the probability 0 and increases exponentially up to the version-level flakiness probability. Flakiness is concentrated in the later runs, with an accelerating increase.
Exponentially Decreasing	The initial flakiness probability of every flaky test decreases exponentially across versions.	The flakiness probability starts from the version-level flakiness probability and decreases exponentially to 0 probability. Flakiness is concentrated in earlier runs, with an accelerating decrease.
Suddenly Increasing	The initial flakiness probability of every flaky test increases by the δ value after a specific version is reached.	All flakiness occurs in a window after a specific run is passed. The flakiness probabilities are set to the probability 0 before the specific run and then the version-level flakiness probability is assigned.
Suddenly Decreasing	The initial flakiness probability of every flaky test decreases by the δ value after a specific version is reached.	All flakiness occurs in a window before a specific run is passed. The flakiness probabilities are set to the version-level flakiness probability before the specific run, and then the 0 probability is assigned.

5.5.2. Run Trend

While the version trend controls the overall flakiness per version, the run trends in Table 2 determine their temporal distribution within a version. This trend simulates realistic scenarios where test flakiness may not be evenly distributed across the execution timeline.

Decoupling the run trend from the version trend allows independent control over the frequency and time when failures occur. This separation is essential to evaluate the ability of each algorithm to detect both distributed and localized flakiness, revealing the strengths and weaknesses of each algorithm in varying testing scenarios.

Unlike version-level, where the δ value represents the difference between the probabilities of the first and last versions, the run trend scales the run-level flakiness probability between 0 and p_v. This means that the probability at a given run p_r is calculated as:

$$p_r = p_v \cdot \frac{r}{R}$$

where the scaling ratio is derived from R and r, the total run count and the current run number, respectively. The dataset implements the same change patterns: *linear*, *exponential*, and *sudden*. The run-level flakiness probabilities are calculated as:

$$\text{Linear:} \quad p_r = p_v \cdot \pm \frac{r}{R} \tag{4}$$

$$\text{Exponential:} \quad p_r = p_v \cdot \pm \left(\frac{r}{R}\right)^2 \tag{5}$$

$$\text{Sudden:} \quad p_r = \begin{cases} 0, & \frac{r}{R} < \rho_T \\ p_v, & \frac{r}{R} \geq \rho_T \end{cases} \tag{6}$$

where ρ_T is the threshold ratio such that $\rho_T = 0.5$ corresponds to the halfway point in the run sequence.

5.6. Report Presence

Although flakiness scoring provides quantitative values for identifying flaky tests, it does not naturally differentiate between explainable and unexplainable failures. In practical terms, not all failed tests with a high failure count or sporadic occurrences present the same flakiness severity. The availability of an attached failure report represents a documented explanation of why a test failed, which may include information on the root cause, or context possibly reducing the effort required to address the problem and ultimately the severity of flakiness.

Within the framework, this report presence is represented by a report flag attached to each test. When a test is generated, the report flag is also decided independently according to a specified probability.

6. Results

We generate and evaluate a dataset using the framework, available in our replication package [17]. During evaluation, we focus on dataset creation and the extent to which the generated dataset captures the characteristics necessary for a meaningful assessment of the two exemplary scoring algorithms, the No Fault Found (NFF) algorithm by Rehman et al. [16] and the Extended Flakiness Score (EFS) by Kisaakye et al. [12]. This includes demonstrating how the dataset meets the established criteria for a "good" dataset, by examining the distribution of result and report associations, the diversity of flakiness patterns, and how the two algorithms would "observe" the dataset. These characteristics are demonstrated using graphs of version trends and run trends, as well as visual representations of flakiness probabilities of the ground truth of a single suite. This suite selected is the one with the trend pair of increasing version flakiness and decreasing run flakiness, which will be called the **increase-decrease suite** for simplicity throughout the rest of this section. The example suite is one of the 49 generated suites within the dataset, chosen because it tries to portray a realistic development scenario: A situation in which overall flakiness increases with each version, due to the growing complexity of the software, meanwhile the development team continuously works on improving the system stability and fixing flaky tests during a version so the run-level flakiness gradually decreases.

6.1. Dataset Overview

By iterating through all the combinations of the version and run trends discussed in Section 5, the framework creates a dataset with 49 unique test suites, each modeling a distinct flakiness pattern simulated over a year.

Each suite consists of; **100 tests** each simulated with **4 versions** with **250 runs** in each version. This yields a total of **5,000,000 result entries** for investigation.

Each test run in the dataset is represented by the following fields, ensuring compatibility with different flakiness scoring models:

- `Test ID`, `Release ID`, and `Run ID` to support detailed tracking of each test result.
- `Report Flag` indicating whether a report was created for the test.
- `Verdict` One of (`Successful`, `Fail`, `Error`, `Skip`) indicating the actual result.
- `Execution Timestamp`, representing the time at which the test was executed.

6.1.1. Test-Level Flakiness Assignment

Figure 1 presents the average base flakiness probability assigned to each test in the selected increase–decrease suite, sorted in ascending order. This base probability is not the final flakiness, but rather serves as an initial parameter from which the flakiness for each version will be derived according to the assigned version trend.

In this configuration, the flakiness probabilities assigned at the test level for flaky tests range between 10% and 40%. The distribution of test categories is 20% clear tests, 20% faulty tests, and 60% flaky tests, reflecting a scenario in which the majority of the test suite is flaky to varying degrees.

6.1.2. Version-Level Flakiness Assignment

Table 3 describes, and Figure 2 illustrates each of the seven version trends by their version-level flakiness probability averages.

Table 3
Version-Level Flakiness Trend Types

Trend	Summary	Equation	Figure
Uniform	Flakiness remains constant across versions.	—	2g
Decrease	Flakiness changes linearly across versions *with a negative slope*.	1	2a
Increase	Flakiness changes linearly across versions *with a positive slope*.	1	2c
Decrease Exponential	Flakiness changes exponentially across versions *with a negative rate* (slow early decrease, faster later).	2	2b
Increase Exponential	Flakiness changes exponentially across versions *with a positive rate* (slow early increase, faster later).	2	2d
Sudden Decrease	Flakiness changes suddenly at threshold version T *by a negative step*.	3	2e
Sudden Increase	Flakiness changes suddenly at threshold version T *by a positive step*.	3	2f

6.1.3. Run-Level Flakiness Assignment

The Table 4 describes and Figure 3 illustrates each of the seven run trends by their run-level flakiness probability averages.

Table 4
Run-Level Flakiness Trend Types

Trend	Summary	Equation	Figure
Uniform	Flakiness remains constant across runs.	—	3g
Decrease	Flakiness changes linearly across runs *with a negative slope*.	4	3a
Increase	Flakiness changes linearly across runs *with a positive slope*.	4	3c
Decrease Exponential	Flakiness changes exponentially across runs *with a negative rate* (slow early decrease, faster later).	5	3b
Increase Exponential	Flakiness changes exponentially across runs *with a positive rate* (slow early increase, faster later).	5	3d
Sudden Decrease	Flakiness changes abruptly at threshold run T_r *by a negative step*.	6	3e
Sudden Increase	Flakiness changes abruptly at threshold run T_r *by a positive step*.	6	3f

6.2. Assumptions on Flakiness Distribution

To ensure a fair, yet challenging, evaluation setting, the probabilities were carefully chosen. 20% are designated as *clear*, always passing, and another 20% as *faulty*, always failing. This guaranties that the presence of non-flaky tests, existing in equal proportions within the dataset, providing an equal baseline by which flakiness can be distinguished. The remaining 60% of the tests are *flaky*. Each *flaky* test was assigned a base flakiness probability drawn uniformly from the range of 0.1 to 0.4. This uniform assignment ensures diversity in flakiness severity while avoiding bias toward particular instability levels.

When generating each individual run, if the execution was *flaky*, its outcome was randomly determined according to additional probability weights. Reports were attached in only 20% of such *flaky* runs, simulating the noise of a system. *flaky* results were assigned: skip (10%), error (10%), and fail (80%). This selection ensures failures are the dominant, while still providing a variety of result states.

This probability design was motivated by two goals:

- Setting 60% of the dataset as *flaky*, slightly more than half of all tests, provides sufficient coverage for evaluation while preserving a significant share of deterministic results.
- The distribution of the presence of the report (20%) was aligned with the combined proportion of skip and error results (10% + 10%). In this way, both algorithms face an equal share of *non-fail* result states.

6.3. Algorithm Perspectives

In this section, we examine how the NFF and EFS algorithms interpret the artificially generated dataset, focusing on the increase-decrease suite as the representative example.

6.3.1. Trend Correlation with Ground Truth

Figure 4a and Figure 4c present the average NFF Rate computed in the example suite, allowing a direct comparison with the ground truth trends shown in Figure 4b and Figure 4d. The version-level analysis in Figure 4a shows an increasing average of the NFF Rate over successive versions. This pattern is consistent with the expectation from ground truth as the system evolves and its complexity increases. The run-level analysis in Figure 4c reveals a decreasing trend in the average NFF rates on run basis. This reflects the ongoing stabilization efforts during the version, aligning with the ground truth towards the later runs.

The alignment between these NFF Rate patterns and the ground truth demonstrates that the intended version and run trends are preserved in NFF algorithm-specific metrics. This outcome confirms that the NFF algorithm successfully captures the underlying flakiness dynamics through its own metric definitions.

6.3.2. Outcome Correlation of NFF and EFS

Figure 5a and Figure 5b present the outcome ratios according to the assumptions embedded within the NFF and EFS algorithms. When these figures are compared, both algorithms are observed to capture the intended outcome composition of approximately 20% *clear*, 20% *faulty*, and 60% *flaky* tests.

The perspective of the EFS algorithm decomposes all possible outcomes *successful*, *skip*, *error*, and *fail*. The NFF algorithm does not differentiate between *clear* and *faulty* tests; all tests rather than failures without a report are visualized in green.

When comparing the two plots, a strong correlation is observed in the maximum and minimum ratios of flaky results. Both algorithms consider the flakiest tests at around 20% flaky result ratio across all runs, with the remaining flaky tests showing a gradual decrease over the suite.

The slight difference in the ratio between the NFF and EFS plots originates from the recognition of skip (yellow), error (purple) states, and faulty tests (full red) by the EFS algorithm that are not considered

individually by the NFF algorithm. In addition to this visual difference, both graphs correlate with each other in outcome variance and distribution in terms of providing a fair comparison ground for both algorithms.

6.3.3. History Correlation of NFF and EFS

Figure 6a and Figure 6b present the test execution results for the first 20 tests of the last version of the example suite. These histories visualize how each algorithm observes run-level flakiness according to its mathematical definition of flakiness. NFF looks at report presence, while EFS looks at test outcomes directly. Despite the definitional differences of flakiness, both algorithms exhibit strong alignment in the positional distribution of flaky outcomes such that the same runs are generally marked as flaky in both histories. This alignment demonstrates that the dataset satisfies run-level compatibility for algorithm-agnostic comparisons.

Another notable observation is the visible influence of the decreasing run trend on the positioning of the flaky outcomes. It can be observed that flaky outcomes are concentrated at the start of the version. This pattern is consistent across both algorithms, further validating that the simulated dataset accurately embeds the intended run trend characteristics.

6.4. Summary of Dataset Generation Results

The results presented in this section demonstrate that the artificially generated dataset successfully embeds the intended version and runs trends while maintaining algorithm-natural compatibility. By configuring diverse combinations with version-level and run-level flakiness probability distributions, the dataset captures a wide range of possible software testing scenarios. The ground truth plots confirm these patterns from the perspective of the target algorithms. The functional requirements of each algorithm are satisfied by the generated dataset on report flags, result variety, and varying distributions of flakiness.

Furthermore, algorithm-specific analyses for NFF and EFS show that both algorithms are able to detect trends within the underlying dataset configuration, despite differences in their definitions of flakiness. The close correlation between algorithm-specific metrics and definitions validates the integrity of the dataset and ensures that the comparative analysis between NFF and EFS can be performed on a fair and representative basis.

7. Related Work

The International Dataset of Flaky Tests (IDoFT) is a collection of flaky tests in Java and Python represented as project URLs along with identifying features such as the commit when flakiness was detected, module path, fully qualified test name, category, and status [18]. While this dataset is invaluable to test flakiness research, the absence of actual test results hinders it's usability as a benchmark for the comparison of different flakiness scoring algorithms.

To address this gap, Wendler and Winter published a regression test history dataset to aid test flakiness research [19]. Their dataset features eleven module-flakiness introducing commit combinations from 8 Maven projects included within the IDoFT dataset. The complete dataset contains 28200 test result histories for 840 tests with history lengths ranging from 1 to 474 commits. This dataset would be an excellent first step toward comparing flakiness scoring algorithms against real data. However, since the underlying probability of flakiness is unknown, it cannot enable a rigorous comparison of flakiness scoring formulae and algorithms under different situations. This is the gap our work seeks to fill.

Regarding dataset generation, one other study comes close to our work. FLAKYRANK is a ranking framework proposed by Wang et al. that relies on augmented learning principles [20]. To address the under-representation of flaky tests in their training dataset, Wang et al. used Generative Adversarial Networks to create synthetic examples. However, the purpose of their dataset generation is different from our work. Their dataset is based on the FlakeFlagger dataset [21]. Therefore, their dataset

generation approach is designed to create a dataset to evaluate approaches that identify flakiness without rerunning. As such, test result and report history are not part of the features in their dataset generation approach.

8. Threats to Validity

One threat concerns the gap between the flakiness of the artificial and real world. In our dataset generation, we proposed a generic way of the data creation process by combining pre-defined trends. Although this approach provides a systematic way to simulate controlled test histories, real-world flakiness features occurring in unique patterns or edge cases may not be captured by our method.

In addition, since we re-implemented the flakiness scoring algorithms, our interpretation may differ from the one used in the original works. However, we do not expect a considerable deviation from the original implementations because we used the same formulae presented in their works.

Finally, the choice of algorithms we implemented for this work, while representative, presents a threat since we do not capture any edge cases that may occur during scoring. However, we do not expect large deviations in the way algorithms relying solely on test execution history will observe the generated dataset.

9. Conclusion

Recognising the role of flakiness scores as decision support during the process of flakiness mitigation, this work examined the different flakiness scoring algorithms proposed in the white and grey literature. We identified a gap in the literature pertaining to a mechanism through which to generate reproducible datasets to rigorously assess the strengths and weakness of different flakiness scoring algorithms. To address this gap, we developed an algorithm-neutral dataset generation framework (FlaDaGe) that can be used to model different flakiness situations and assess different flakiness scoring algorithms. Finally, we demonstrated how it can be used to generate a dataset to assess the performance of two pre-existing flakiness scoring algorithms defined in [16] and [12], showing how each algorithm "observes" the simulated flakiness. In future work, the dataset generation framework can be extended to support a wider range of features necessary for new flakiness scoring algorithms. A broader evaluation against the rest of the algortihms, and a comparison of their performance against real world data, may also be conducted.

Acknowledgments

This work is supported by the Research Foundation Flanders (FWO) via the BaseCamp Zero Project under Grant number S000323N.

References

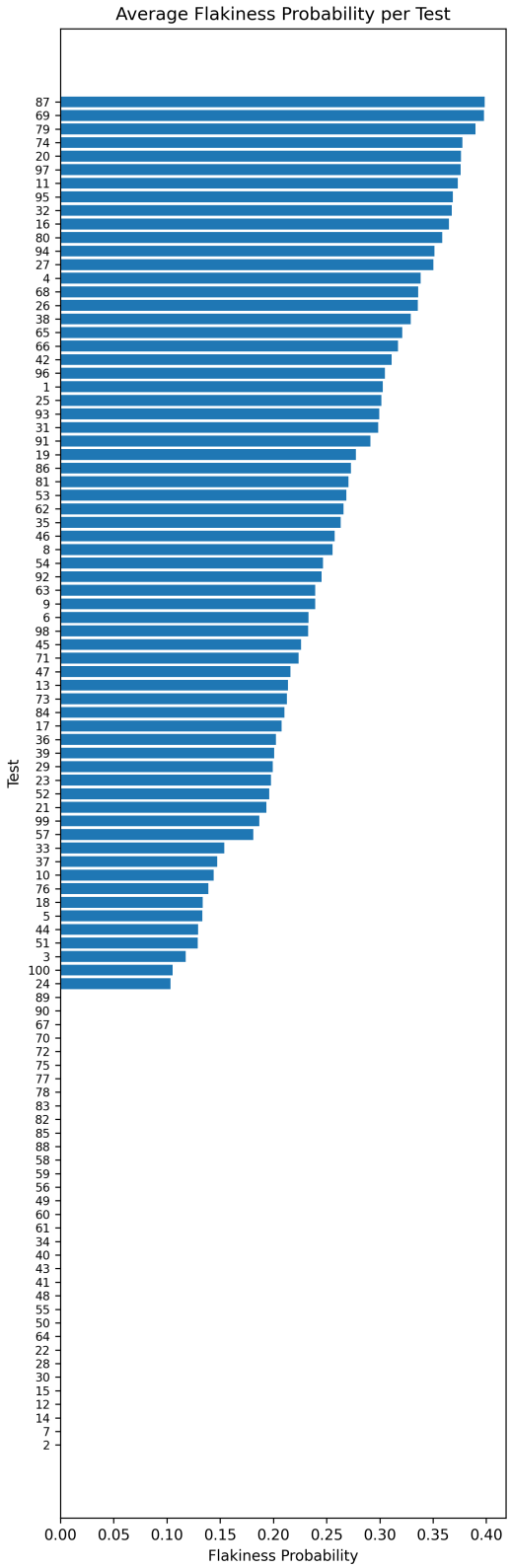
[1] F. Lonetti, E. Marchetti, Emerging software testing technologies, in: Advances in computers, volume 108, Elsevier, 2018, pp. 91–143.
[2] A. Bertolino, Software testing, SWEBOK 69 (2001).
[3] O. Parry, G. M. Kapfhammer, M. Hilton, P. McMinn, A survey of flaky tests, ACM Transactions on Software Engineering and Methodology (TOSEM) 31 (2021) 1–74.
[4] A. Tahir, S. Rasheed, J. Dietrich, N. Hashemi, L. Zhang, Test flakiness' causes, detection, impact and responses: A multivocal review, Journal of Systems and Software 206 (2023) 111837.
[5] Q. Luo, F. Hariri, L. Eloussi, D. Marinov, An empirical analysis of flaky tests, in: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, 2014, pp. 643–653.

[6] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, J. Bell, A large-scale longitudinal study of flaky tests, Proceedings of the ACM on Programming Languages 4 (2020) 1–29.
[7] W. Lam, K. Muşlu, H. Sajnani, S. Thummalapenta, A study on the lifecycle of flaky tests, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 1471–1482.
[8] A. Berndt, S. Baltes, T. Bach, Taming timeout flakiness: An empirical study of sap hana, in: Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice, 2024, pp. 69–80.
[9] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, A. Memon, Modeling and ranking flaky tests at apple, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice, 2020, pp. 110–119.
[10] M. Gruber, M. Heine, N. Oster, M. Philippsen, G. Fraser, Practical Flaky Test Prediction using Common Code Evolution and Test History Data , in: 2023 IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE Computer Society, Los Alamitos, CA, USA, 2023, pp. 210–221. URL: https://doi.ieeecomputersociety.org/10.1109/ICST57152.2023.00028. doi:10.1109/ICST57152.2023.00028.
[11] M. H. U. Rehman, P. C. Rigby, Quantifying no-fault-found test failures to prioritize inspection of flaky tests at ericsson, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 1371–1380.
[12] J. Kisaakye, M. Beyazıt, S. Demeyer, Extending a flakiness score for system-level tests, in: IFIP International Conference on Testing Software and Systems, Springer, 2024, pp. 292–312.
[13] Meta Engineering Team, How do you test your tests? A Probabilistic Flakiness Score for Testing at Scale, 2020. URL: https://engineering.fb.com/2020/12/10/developer-tools/probabilistic-flakiness/.
[14] S. Rasheed, J. Dietrich, A. Tahir, On the Effect of Instrumentation on Test Flakiness , in: 2023 IEEE/ACM International Conference on Automation of Software Test (AST), IEEE Computer Society, Los Alamitos, CA, USA, 2023, pp. 123–127. URL: https://doi.ieeecomputersociety.org/10.1109/AST58925.2023.00016. doi:10.1109/AST58925.2023.00016.
[15] G. Haben, S. Habchi, J. Micco, M. Harman, M. Papadakis, M. Cordy, Y. Le Traon, The importance of accounting for execution failures when predicting test flakiness, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 1979–1989. URL: https://doi.org/10.1145/3691620.3695261. doi:10.1145/3691620.3695261.
[16] M. H. U. Rehman, Quantifying Flaky Tests to Detect Test Instabilities, Ph.D. thesis, Master's thesis. Concordia University. https://spectrum. library. concordia …, 2019.
[17] Anonymous, Fladage: A framework for generation of synthetic data to compare flakiness scores, 2025. URL: https://doi.org/10.5281/zenodo.17206909. doi:10.5281/zenodo.17206909.
[18] W. Lam, International Dataset of Flaky Tests (IDoFT), 2020. URL: http://mir.cs.illinois.edu/flakytests.
[19] P. Wendler, S. Winter, Regression-test history data for flaky-test research, in: Proceedings of the 1st International Workshop on Flaky Tests, FTW '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 3–4. URL: https://doi.org/10.1145/3643656.3643901. doi:10.1145/3643656.3643901.
[20] J. Wang, Y. Lei, M. Li, G. Ren, H. Xie, S. Jin, J. Li, J. Hu, Flakyrank: Predicting flaky tests using augmented learning to rank, in: 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2024, pp. 872–883.
[21] A. Alshammari, C. Morris, M. Hilton, J. Bell, Flakeflagger: Predicting flakiness without rerunning tests, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 1572–1584. doi:10.1109/ICSE43902.2021.00140.

A. Flakiness Assignment

A.1. Test-Level Flakiness Assignment

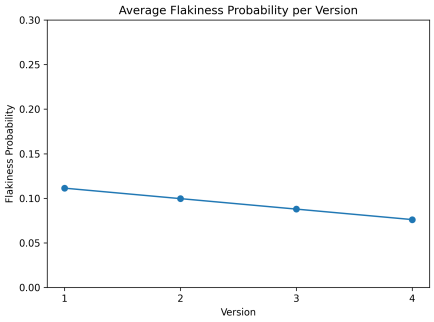
Figure 1: Average Test-Level Flakiness Probability per Test



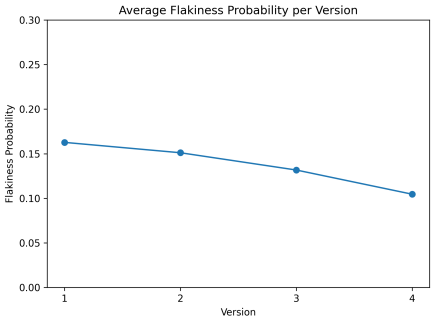
A.2. Version-Level Flakiness Assignment

Figure 2: Version-Level Flakiness Probability Averages

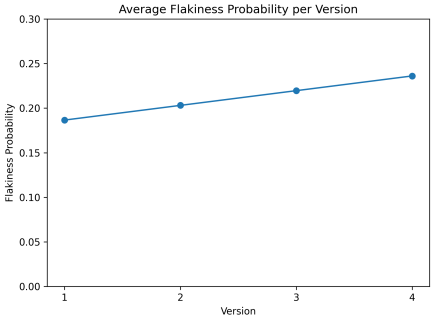
(a) Decrease



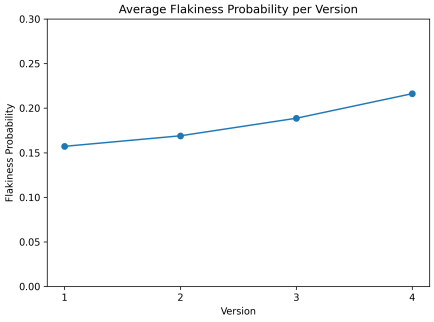
(b) Decrease exponential



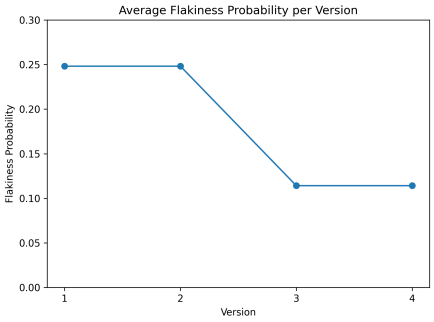
(c) Increase



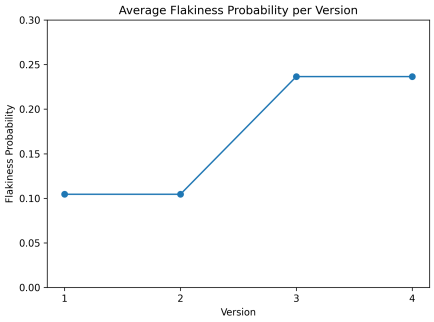
(d) Increase exponential



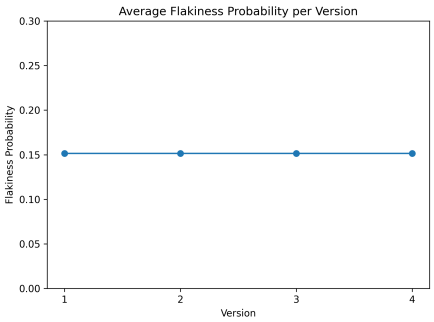
(e) Sudden decrease



(f) Sudden increase



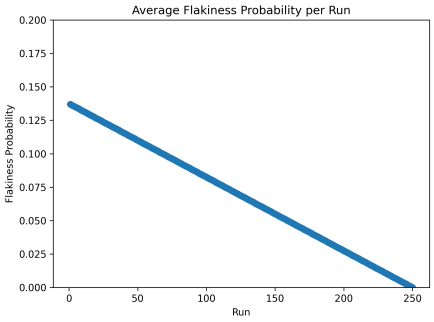
(g) Uniform



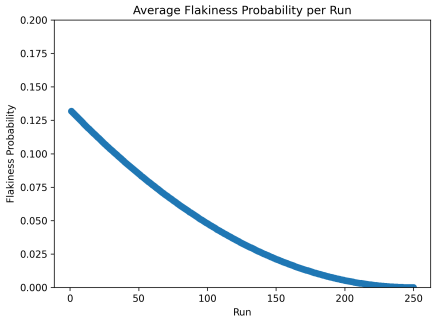
A.3. Run-Level Flakiness Assignment

Figure 3: Run-Level Flakiness Probability Averages

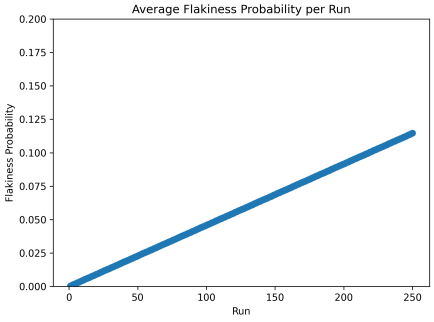
(a) Decrease



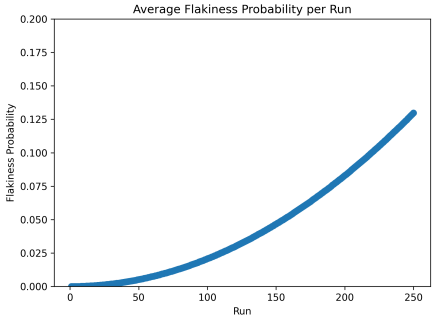
(b) Decrease exponential



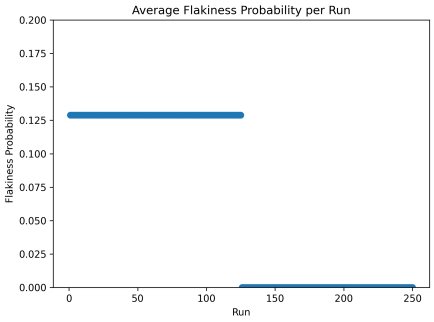
(c) Increase



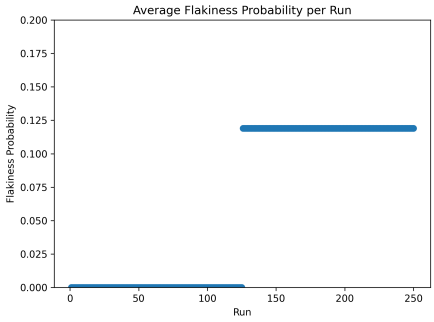
(d) Increase exponential



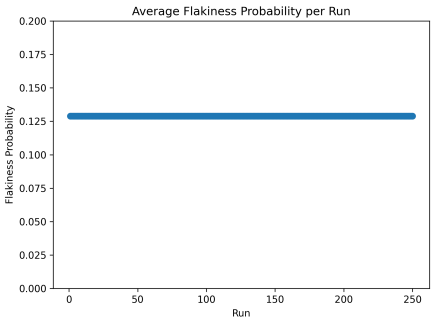
(e) Sudden decrease



(f) Sudden increase



(g) Uniform

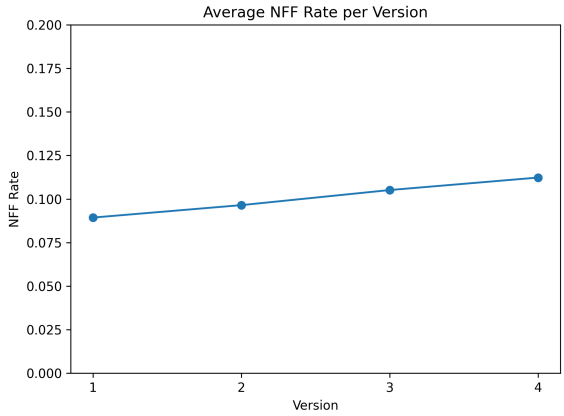


B. Algorithm Perspectives

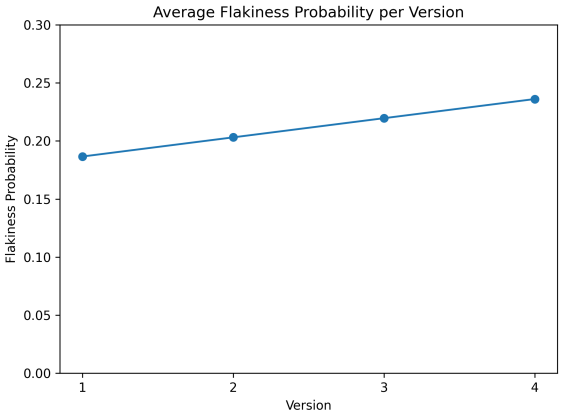
B.1. Trend Correlation with Ground Truth

Figure 4: NFF and Ground Truth Pattern Correlation

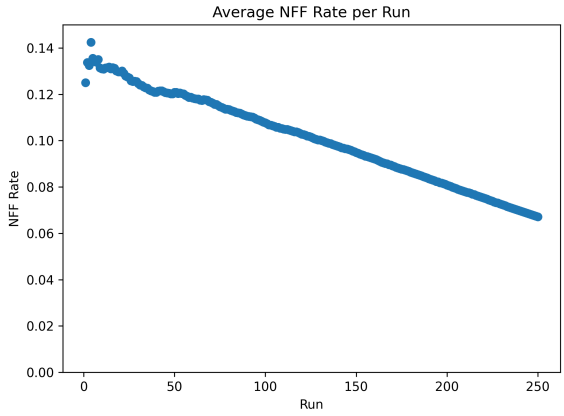
(a) Average NFF rate per version



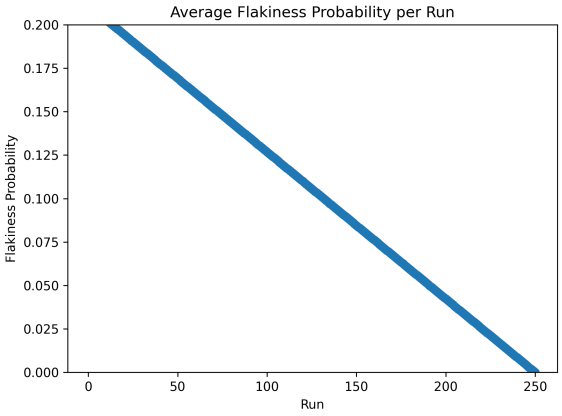
(b) Average flakiness probability per version



(c) Average NFF rate per run



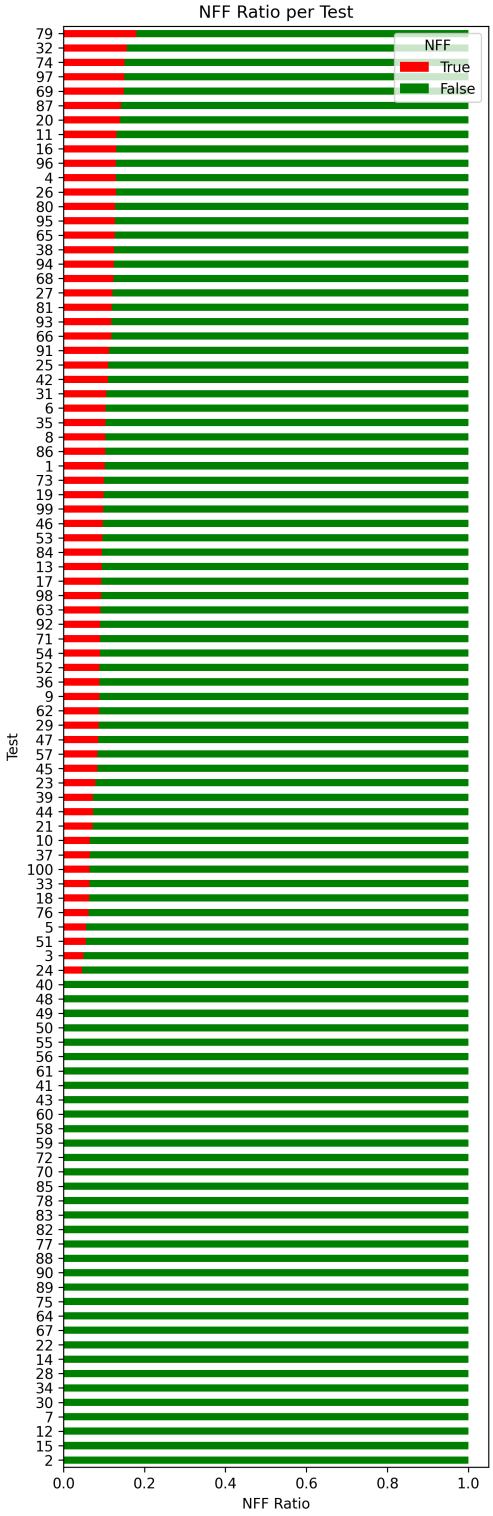
(d) Average flakiness probability per run



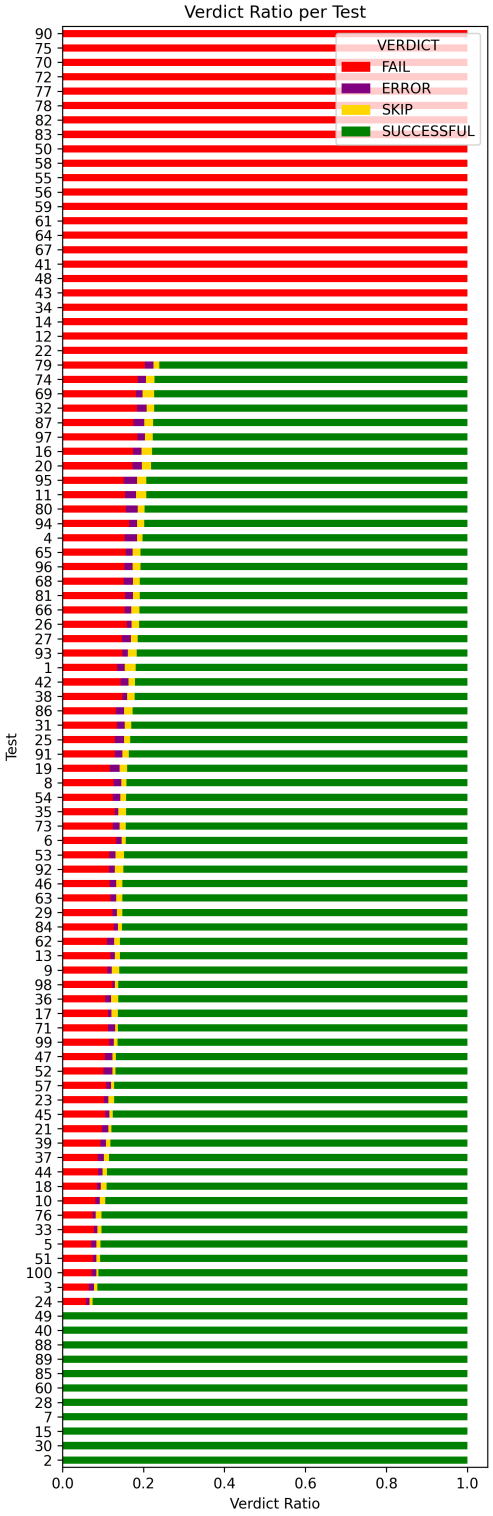
B.2. Outcome Correlation of NFF and EFS

Figure 5: Comparison of NFF and EFS Ratio per Test

(a) NFF ratio per test



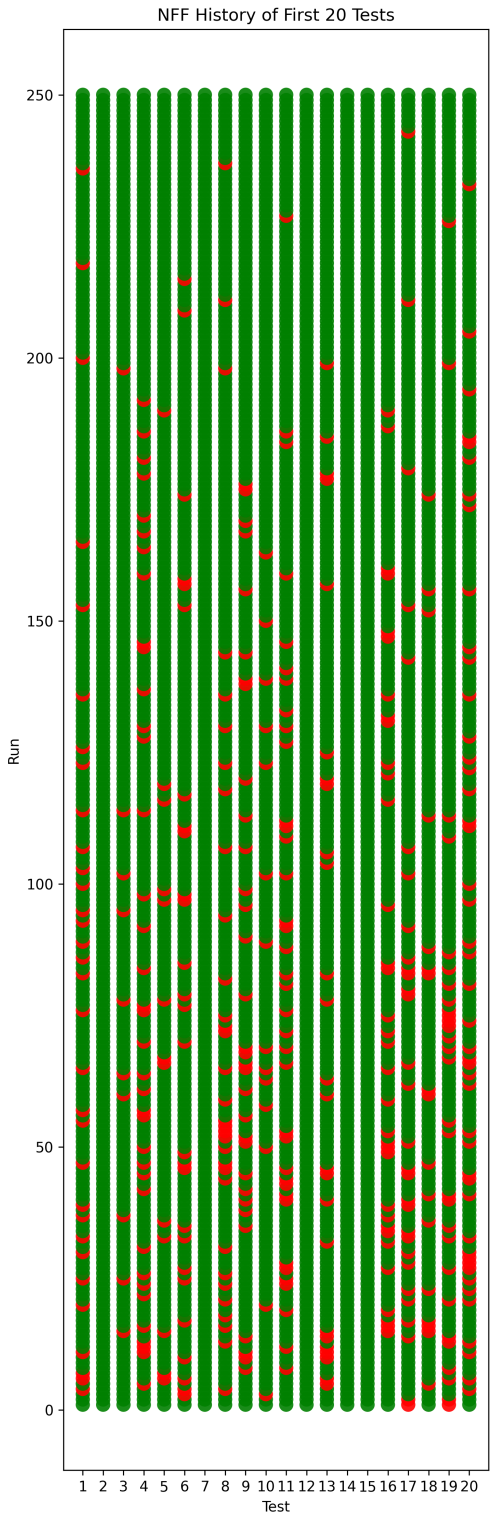
(b) EFS result ratio per test



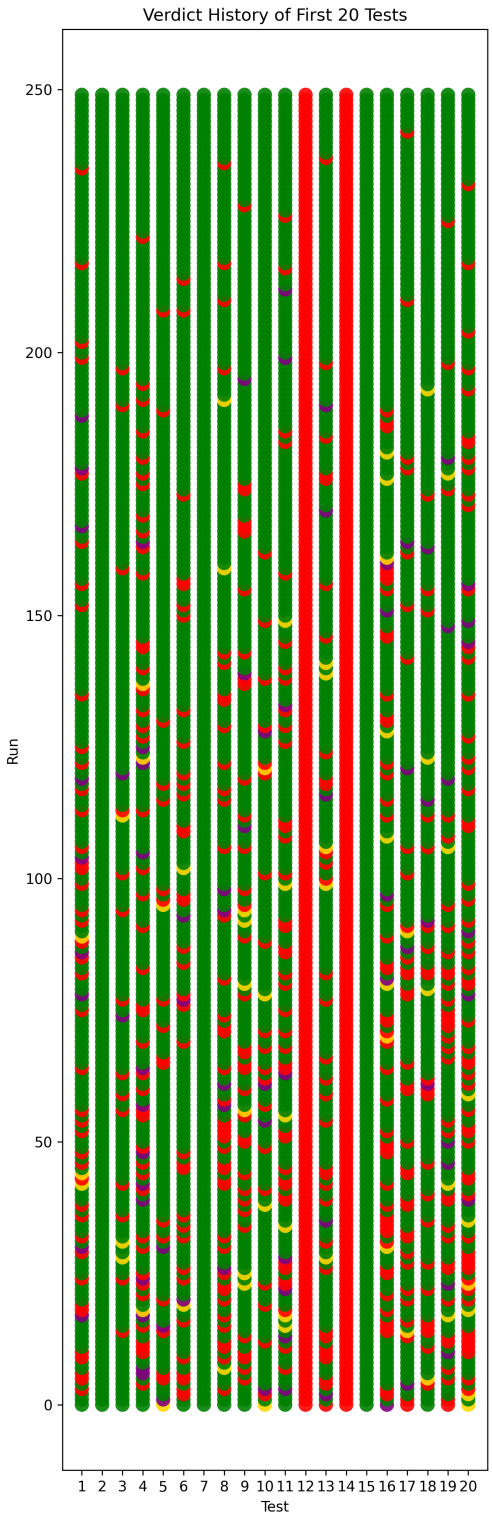
B.3. History Correlation of NFF and EFS

Figure 6: Comparison of NFF and EFS Views of History for the First 20 Tests

(a) NFF View of History of First 20 Tests



(b) EFS View of History of First 20 Tests



Evaluating Test-Driven Code Generation: A Replication Study

Giovanni Rosa*, Jesus M. Gonzalez-Barahona*

SoftDev group, Universidad Rey Juan Carlos, Spain

Abstract

The software engineering community is exploring ways of integrating Large Language Models (LLMs) into software developing processes. One of such explorations is the use of techniques based on Test-Driven Development (TDD), which presents significant challenges due to model variability, evolving APIs, and computational resource demands. In this case, the interest is usually not in showing how well a model behaves, but how a technique may improve the results of the LLM in certain scenarios.

This paper presents a replication study of one of those explorations: *AlphaCodium*, an approach implementing a TDD-based workflow for producing code with LLMs, evaluated by solving competitive programming problems from the *CodeContests* dataset. The primary goal of our study is not the replication itself, but to understand how to empirically evaluate LLM-based code generation approaches, and the challenges involved in replicating such experiments. However, our replication also aims to add more evidence about the effectiveness of the approach.

We extended the original open-source *AlphaCodium* implementation to support some open-weights models, replicating the evaluation on the same dataset. In the process, we collected some supplementary inference metrics such as latency and token usage.

By discussing the outcome and challenges encountered, we offer a set of actionable takeaways that can help future replication studies of multi-step approaches for LLM-based code generation.

Keywords

code generation, TDD, LLM, replication study, performance evaluation, software engineering

1. Introduction

Large Language Models (LLMs) have shown remarkable capabilities in various natural language processing tasks, and have been widely adopted in the software engineering domain [1]. Starting from code assistants to help developers write, debug, and refactor code [2], the development of code-specific LLMs has been a major focus in recent years [3, 4]. LLMs have been employed to tackle competitive programming problems, where the models are measured by their capability of generating code following specific functional requirements and passing a set of test cases. An important mention is *AlphaCode* [5], developed by Google DeepMind, which demonstrated effective in that purpose.

The software engineering community is exploring ways of integrating LLMs into software development processes, with different purposes: to produce better code, to collaborate in software maintenance tasks such as bug fixing, or to improve how they take into account contextual information. One of these explorations deals with the use of Test Driven Development (TDD) [6, 7] to improve code generation. The general idea in this realm is to make LLMs iteratively generate and refine code based on a provided set of test cases, which are expected to guide the model towards producing correct and robust implementations, given a specification. These approaches are usually evaluated by trying the approach to solve coding problems for which tests can be found.

However, evaluating and replicating any kind of experiment involving LLMs can be challenging due to the rapid evolution of the models, the inherent variability in LLM outputs, which can lead to inconsistent results across different runs, and the fair evaluation of the approaches relying on LLMs, which can be difficult due to the high costs associated with using proprietary models and the complexity

Benevol'25: 24th Belgium-Netherlands Software Evolution Workshop 17–18 November 2025, Enschede, The Netherlands

*Corresponding authors.

giovanni.rosa@urjc.es (G. Rosa); jesus.gonzalez.barahona@urjc.es (J. M. Gonzalez-Barahona)

https://giovannirosa.com (G. Rosa); https://gsyc.urjc.es/jgb/ (J. M. Gonzalez-Barahona)

0000-0002-5241-1608 (G. Rosa); 0000-0001-9682-460X (J. M. Gonzalez-Barahona)

of setting up open-source models. This is even more challenging when the aim is not to evaluate the model itself, but how a given technique (*i.e.,* derivative of TDD) may improve the performance of LLMs.

In this paper, we present a replication study designed to understand how to empirically evaluate LLM-based code generation approaches, and the challenges involved in replicating such experiments. To this aim, we replicate the study of Ridnik *et al.* [8] which introduced *AlphaCodium*, a TDD-based approach for enhancing the effectiveness of LLMs in code generation. The approach involves a robust pre-processing phase, including problem reflection and test case augmentation, which extends similar approaches in the literature [6, 7], and most importantly, the source code is publicly available on GitHub [9], facilitating the replication process. We evaluate the approach on the *CodeContests* dataset [5], composed of coding problems, which is the same used by the original study, demonstrating significant improvements over a simple direct code generation method.

Note that in this kind of evaluation, the focus is not on the capabilities of the LLMs themselves, but rather how certain techniques, usually related to software engineering practices, can be used to improve results in certain tasks. Given this situation, our goal is to understand the challenges related to the replication and evaluation of the complex workflow implemented in *AlphaCodium*, rather than the performance of the LLMs themselves.

The authors of the original study provided an open-source implementation of *AlphaCodium*, which we used as a starting point for our study. We leveraged this implementation by extending and modifying it to support additional LLMs, such as open-weights models, and to extend the evaluation parameters. Using our tool, we could replicate the original study with other LLMs, and report on some other performance-related parameters not considered in it, such as latency and token usage. We report the results of our replication study and provide some insights into the challenges and common errors that arose during the replication process, along with a set of takeaways for future replication studies.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 describes the experimental protocol of the replication study. We present the results in Section 4, while in Section 5 we discuss what we learned during the replication process. Finally, in Section 6 we report the threats of validity for the study and Section 7 concludes the paper.

2. Related Work

Large Language Models (LLMs) have been used extensively in code-related tasks, such as code completion and code generation. A seminal study in this field introduced the *Codex* models and evaluated them on competitive programming problems [10]. They introduced the `pass@k` metric, which is now widely used to evaluate code generation models. Several studies have conducted similar experiments to produce models and approaches specialized in solving coding problems. For example, Google Deepmind introduced *AlphaCode* [5], improved in *AlphaCode 2* [11], which is able to reach a competitive level in programming competitions.

Exploring ways of combining LLMs with software engineering practices aimed at improving code quality, several studies investigated the usage of LLMs to solve coding problems with the aid of Test Driven Development (TDD) practices. Piya *et al.* [6, 12] investigated the impact of TDD practices on the performance of LLMs in solving *LeetCode* problems, showing that TDD can significantly improve the performance of LLMs in solving coding problems. Fakhoury *et al.* [13] introduced TICODER, a workflow for test-driven, interactive code generation. TICODER automatically generates tests and code candidates from user requirements, allowing iterative refinement through user feedback. Their evaluation, using both user studies and benchmark datasets such as *MBPP* [14] and *HumanEval* [10], demonstrates that test cases can effectively enhance code quality. Similarly, Mathews *et al.* [7] proposed TGen, which applies TDD principles to LLM-based code generation. Starting from input test cases, TGen generates code and iteratively remediates it based on failed test outputs until all tests pass. Experiments on MBPP and HumanEval show that TDD-based workflows significantly improve the correctness and robustness of LLM-generated code.

AlphaCodium [8] proposes a similar technique, introducing a robust pre-processing phase including a

problem reflection step and a test cases augmentation procedure. The authors evaluated their approach on the *CodeContests* dataset, showing that it significantly improves the performance of LLMs in solving competitive programming problems. We selected *AlphaCodium* as the replication target for our study, leveraging the open implementation provided by its authors [9]. We describe it more in detail in Section 3.1.

3. Replication Study Design

We report a replication study of the paper of Ridnik *et al.* [8], with the *goal* of understanding how to evaluate approaches built upon LLMs and the challenges faced when dealing with LLM-based experiments designed not to evaluate the models themselves, but relatively complex pipelines using them.

Our replication study is performed to answer the following *research question*:

> *RQ: To what extent are we able to replicate the empirical evaluation of the AlphaCodium approach?*

We aim to replicate the results of the original paper as closely as possible, focusing on how to evaluate approaches built upon LLMs and the related challenges when dealing with LLM-based experiments.

3.1. The AlphaCodium Approach

The approach presented in the original study we replicate, called *AlphaCodium*, is based on the idea that generating code in multiple steps, with intermediate verification and refinement, can lead to better results than a single-step generation. The overall approach is shown in Fig. 1, and can be summarized as follows: Taking as input a coding problem description (specification) and the test cases to verify the solution, *AlphaCodium* (i) generates a possible resolution procedure (*i.e.,* list of steps to solve the problem), (ii) generates a set of possible solutions and selects the best one, (iii) generates additional test cases, and (iv) executes a code generation loop running the test cases and fixing the function implementation. For additional details on the approach, we refer the reader to the original paper [8].

The original study conducted an empirical evaluation of the approach using different LLMs, including open-weights models (*DeepSeek*) and closed ones, like OpenAI's *GPT-3.5* and *GPT-4*. The baseline is composed of a single-step code generation using the same models, using the Zero-shot prompting technique [15], which is basically the simplest way of prompting an LLM to generate code from a specification. Therefore, this is a good example of an evaluation not aimed at evaluating the models themselves, but how they can be improved by using some specific technique. Moreover, the original study compared the results with *AlphaCode* [5], a state-of-the-art approach for code generation based on

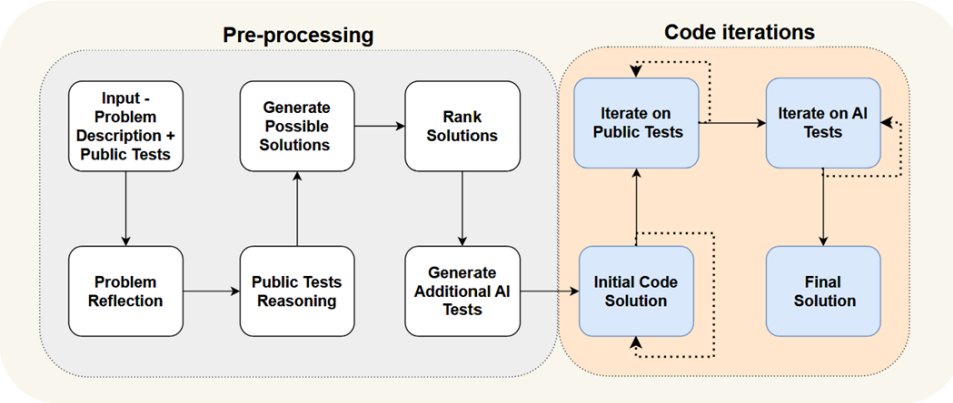


Figure 1: The *AlphaCodium* approach proposed by [8].

LLMs. We selected *AlphaCodium* because we were interested in existing approaches for code generation based on an iterative workflow, combined with TDD, and also because the source code of the tool is publicly available [9].

3.2. Study Context

The *context* of the original study consists of the *CodeContests* dataset [5], a set of coding problems extracted from several programming platforms, such as *Codeforces*[1] and *Codechef*[2]. Each instance of the dataset contains the data to train and evaluate code generation models, including: (i) the problem description, (ii) the function signature, (iii) the public and private test cases, and the (iv) solution code. The full list of fields is summarized on the dataset card[3].

We started with the exact same dataset used in the original paper, publicly available on HuggingFace [4]. While the dataset is composed of training, validation, and test splits, we only use the *validation* and *test* splits, as did the *AlphaCodium* paper, for a total of 117 and 165 instances, respectively. They discarded a total of 12 instances from the test set, resulting in a total of 156 instances, and 9 instances from the validation set, resulting in a total of 105 instances. We investigated the reason behind this, and we found in the code that if less than 20% of the solutions provided in the dataset are correct, the problem is marked as invalid, and then skipped by the tool. Approximately, the knowledge cut-off date for the problems contained in the dataset is around 2021.

3.3. Experimental Procedure

In this section, we describe the experimental procedure we followed to replicate the results of the *AlphaCodium* approach, describing the differences with the original experiment.

3.3.1. Tool implementation

The *AlphaCodium* tool is implemented in Python and is publicly available on GitHub [9], which served as our starting point for the code used for replicating the experiment. One of the main differences in our replication is the supported LLMs. While the original paper uses OpenAI and DeepSeek models, we extended the tool to support open-weights models deployed via *vLLM*[5] and models accessible through *OpenRouter APIs*[6]. We chose vLLM because it allows deploying HuggingFace models with production-ready performance and improved reliability, compared to alternatives like *Ollama*[7]. OpenRouter was selected for the wide availability of model providers, including OpenAI and Anthropic, and a unified pay-per-use API.

Moreover, we made several improvements and fixes to enhance the tool's reliability and robustness. These include better output parsing (handling corner cases), improved exception handling, extended logging (*e.g.,* token usage and latency), and the support for additional configuration parameters. An example is the option to run a single problem instance or a subset of instances, which facilitates debugging and targeted re-execution in case of failures or interruptions. Also, the problem iterations are configured to stop once a valid solution was found. We modified this behavior to always perform the maximum number of iterations, to obtain a complete set of results of the approach effectiveness.

We implemented the storage of intermediate results for each iteration. This allows us to resume the execution from the last completed iteration after interruptions. Also, we set an empty result entry for the cases where the model fails to generate a solution or the generated solution does not pass all the test cases within the maximum number of iterations. This because by default the tool will

[1] https://codeforces.com/
[2] https://www.codechef.com/
[3] https://huggingface.co/datasets/deepmind/code_contests
[4] https://huggingface.co/datasets/talrid/CodeContests_valid_and_test_AlphaCodium
[5] https://docs.vllm.ai
[6] https://openrouter.ai/
[7] https://ollama.com

re-execute these iterations. These changes allowed us to parallelize the execution, significantly speeding up the experiments since each problem instance is independent. We can conclude that we made mostly engineering improvements while preserving as much as possible the original behavior of the tool.

3.3.2. Experimental parameters

The models used for our replication are the following:

- *openai/gpt-3.5-turbo* via OpenRouter APIs[8]: Since *gpt-3.5* is not available on OpenRouter, we opted for the turbo version. The knowledge cut-off is reported as September 2021.
- *deepseek-ai/deepseek-coder-33b-instruct*[9] [4] via vLLM: The original paper only reports *deepseek-coder-33b* as the model. We opted for the *instruct* version, better suited for instruction-following tasks. The knowledge cut-off is not explicitly reported, but the model was released in November 2023, so we can assume the knowledge cut-off is a few months prior to that date.
- *codellama/CodeLlama-34b-Instruct-hf*[10] [3] via vLLM: Created from Meta's *Llama 2* family of models, it is an open-weights model widely used for code generation tasks [16]. The model has been trained between January and July 2023.

Given the cut-off date of the evaluation dataset, we can assess on the likelihood of contamination: that the models had access to the dataset, including solutions, as a part of their training set. The versions of Deepseek and CodeLlama which we use may have contamination, but GPT-3.5-Turbo should not have it according to the reported knowledge cut-off date. In any case, we do not know how much the contamination may affect the results.

We used the same execution parameters provided in the open source implementation of the tool (*i.e.,* TOML configuration file[11]). We made some modifications, mostly by adding new parameters, while trying to keep them as close as possible to the original ones. Specifically, we kept the same model temperature, which depends on the individual prompt and varies between 0.2 and 0.3. We set the parameter for maximum number of iterations to 5. An iteration corresponds to a single attempt to solve the problem and thus, we need a total of five attempts for each problem to compute the `pass@5` metric. We kept the original parameters for the number of possible solutions generated during the pre-processing step (*i.e.,* 3), the total attempts to generate the initial code solution for the TDD loop (*i.e.,* 8). Also, we kept the total number of feedback iterations in the TDD loop set to 3, meaning that the model has three chances to fix the code based on the failed test cases. Please refer to the original configuration file for the complete list of available parameters.

We changed the execution timeout: While the original tool used a timeout of 90 seconds for inference calls, we increased this to 600 seconds to help with the higher latency of open-weights models. This value is the default for the OpenAI Python SDK. We used the `use_baseline` flag as a parameter to switch between the *AlphaCodium* approach and the single-step code generation baseline, resulting in two different resolution procedures for the experiment.

3.3.3. Evaluation metrics

The original paper uses the `pass@k` metric [10], the reference measure for code generation tasks. The metric is computed as follows: Given a set of n generated solutions samples for a problem, for up to k attempts, the `pass@k` metric measures the probability that at least one of the n samples passes all the test cases. As stated in Section 5.3 of the original paper, *"[...] we perform 15-20 LLM calls per solution, so a pass@5 submission involves 100 LLM calls"*. Therefore, we assume that they generated 5 solutions for each problem. Then, we computed the `pass@5` metric[12] using 5 samples per problem

[8] https://openrouter.ai/models/gpt-3.5-turbo
[9] https://huggingface.co/deepseek-ai/deepseek-coder-33b-instruct
[10] https://huggingface.co/codellama/Codellama-34b-instruct
[11] https://github.com/Codium-ai/AlphaCodium/blob/main/alpha_codium/settings/configuration.toml
[12] By definition, computing `pass@k` requires at least $n \geq k$ samples per task.

Figure 2: Example of the command used to deploy *DeepSeek-Coder* via vLLM.

```
vllm serve deepseek-ai/deepseek-coder-33b-instruct \
--host 0.0.0.0 --port 8000 \
--gpu-memory-utilization 0.95 \
--max-model-len 53700 \
--trust-remote-code \
--tensor-parallel-size 2
```

($n = 5$). Each instance of the dataset, along with public and private test cases, contains an additional set of *generated* test cases, created based on the existing public and private ones. We computed the metric using both sets, as in the original paper, and also report the results using only the private test cases to provide an alternative evaluation perspective that is closer to a real-world scenario for coding challenges. Additionally, we report the collected statistics for each inference call as latency (*i.e.,* time taken to get the response from the model), token usage (*i.e.,* total and completion tokens). We report the average values computed for a single iteration, along with the total number of calls to the model.

3.3.4. Execution environment

We deployed the open-weights models on a multi-GPU shared server with a total of 4 A100 GPUs (80 GB), an AMD EPYC 7313 16-Core and 256 GB of RAM. This allowed us to execute full precision models, without the need for quantization, which could influence the performance of the models. For the experiments using the *OpenRouter* APIs, we used a standard laptop machine with an Intel i7-12700H CPU and 64 GB of RAM. To ensure a clean and isolated execution environment, we ran the tool inside a dedicated Docker container. This enhances reproducibility of the execution environment and minimizes any potential security risks associated with unexpected behavior of the generated code. The tool also incorporates a sandboxing mechanisms limiting memory and potentially dangerous commands (*e.g.,* `rm`). The vLLM instance has been deployed inside a separate Docker container on the same server, using two of the four GPUs with tensor parallelism. We then configured the *AlphaCodium* tool to interact with it via REST APIs. Although the GPUs have sufficient memory to load the models, for *DeepSeek-Coder* we needed to limit the maximum context length of the model. This did not affect our experiments, since the prompts are much smaller than this limit. In Fig. 2 we provide an example of the command used to deploy the model.

4. Replication Study Results

Table 1 presents the `pass@k` scores from our replication study, using $k = 5$ and $n = 5$ samples per problem instance. We compare these results with those reported in the original paper [8]. The outcomes are comparable and generally consistent with the original scores.

For the baseline approach, our pass rates are slightly higher, while for *AlphaCodium* the results are similar. This difference may be attributed to the high variance in results due to the limited number of samples per problem instance, particularly for the baseline, which relies on a single call to the LLM.

CodeLlama remains the worst performing model, but its results are still consistent with the original study, where the *AlphaCodium* approach outperforms direct solving. We do not have a direct comparison for *GPT-4*, the best-performing model in the original paper. However, given the release dates and knowledge cut-offs, it is reasonable to assume that it is more capable than the other models, likely benefiting from newer data, training procedures, and having portions of its training data overlapping with the *CodeContests* dataset.

Table 2 shows the results obtained by measuring the `pass@5` score only on the private test set. We can conclude that having fewer test cases makes the overall score higher, assuming that it is easier to

Table 1
Summary of the results for the `pass@5` score evaluated on *validation* and *test* splits. The symbol † indicates those from the original paper [8].

Model	Validation Set (pass@5)		Test Set (pass@5)	
	Zero-Shot	*AlphaCodium*	*Zero-Shot*	*AlphaCodium*
DeepSeek-33B†	7%	20%	12%	24%
GPT-3.5†	15%	25%	8%	17%
GPT-4†	**19%**	**44%**	12%	**29%**
deepseek-ai/deepseek-coder-33b-instruct	10%	18%	**14%**	21%
openai/gpt-3.5-turbo	11%	22%	10%	21%
codellama/CodeLlama-34b-Instruct-hf	2%	5%	4%	8%

Table 2
Summary of the results for the `pass@5` score evaluated only on the problems' private test set.

Model	Validation Set (pass@5)		Test Set (pass@5)	
	Zero-Shot	*AlphaCodium*	*Zero-Shot*	*AlphaCodium*
deepseek-ai/deepseek-coder-33b-instruct	**20%**	**27%**	**22%**	**23%**
openai/gpt-3.5-turbo	19%	**27%**	13%	19%
codellama/CodeLlama-34b-Instruct-hf	6%	9%	10%	5%

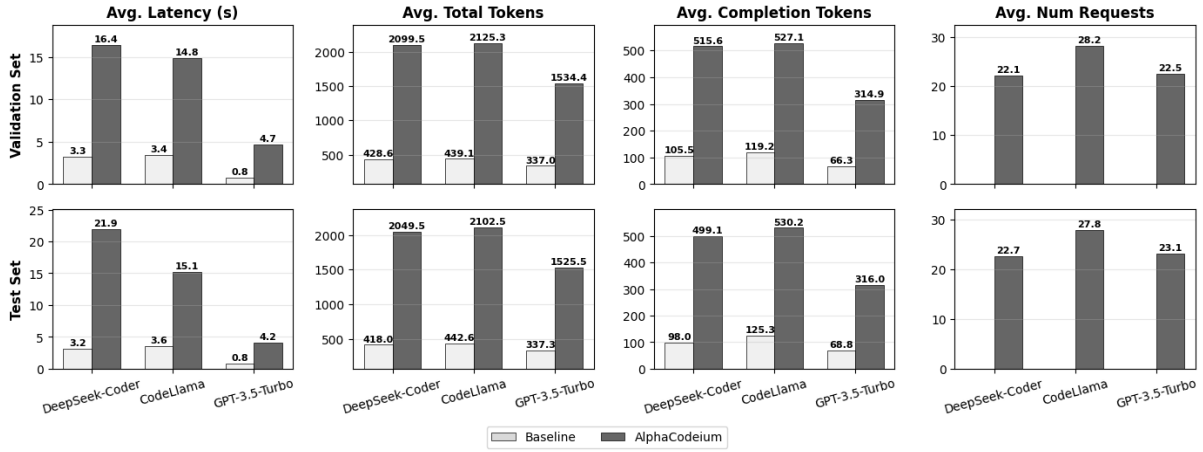


Figure 3: Summary of the inference statistics collected for the experiments running *AlphaCodium*.

produce a solution that passes all the tests. However, the relative performance of the models remains the same, with *AlphaCodium* achieving better results than the baseline approach.

Fig. 3 shows the statistics collected during the model inference. The values represent the average value for a complete execution of a single instance (single iteration). The latency of *GPT-3.5-turbo* is lower than the other models, because of *OpenRouter*. A fairer comparison is between *DeepSeek-Coder-33B* and *CodeLlama-34B*, both deployed on the same hardware, where we can see that the former is much slower than the latter, probably due to the complexity of the model. In terms of token usage, *GPT-3.5-turbo* is more efficient despite performing similarly to *DeepSeek-Coder-33B* in terms of number of requests and achieved pass rate. Additionally, the total token usage for *AlphaCodium* is significantly higher than for the baseline approach, as expected, since it makes multiple requests to the LLM for each problem instance. For a fairer comparison, it would be better to consider the overall cost of each approach, or to allocate the same token budget to both methods and compare their results accordingly.

5. Challenges and Lessons Learned

Replicating and evaluating the experiments proved to be challenging, specifically when dealing with large language models (LLMs) that evolve rapidly. Below, we outline the issues we encountered, our mitigations, and lessons learned.

Flaky formatting errors. The non-determinism of the execution of the models led to flaky errors related to formatting in the generated YAML output. Actually, the prompt requests to generate a YAML-formatted output with the required fields (*e.g.,* test cases, resolution procedure, etc.). However, some models struggled to adhere strictly to this format, leading to parsing errors when the output is processed by a YAML parser. We chose to not force the parsing of that output, considering those cases as invalid responses from the model.

Token generation loop. In our replication, we worked with LLMs that are older and less capable than current ones. A recurrent issue, especially with `deepseek-coder-33b-instruct`, is that the model sometimes enters a token generation loop, repeatedly generating the same or random tokens without making progress toward a valid response. In those cases, the inference exceeds the timeout and the generation stops. Identifying these cases is not trivial. One strategy could be to use streaming output and monitor the generated tokens, although that would make the code more complex. Fortunately, we observed that the average latency for normal responses is much lower than for these generation loops (see Fig. 3). Thus, a simple and effective mitigation is to use an anomaly detection heuristic based on a threshold on average latency to identify these cases, and rerun the inference.

Unable to execute the generated code. Another issue is that, in several cases, the generated code cannot be executed either due to syntax errors or missing dependencies. Even if the *AlphaCodium* pipeline includes an iteration phase with a code-fixing loop, we found that some models struggle to generate syntactically correct code, leading to execution failures. Examples are invalid code indentation, error while parsing the input of test cases, recalls to missing functions or files, accessing invalid list ranges or invalid type comparison (*e.g.,* string with integer). Other frequent errors come from the presence of explanations inside the code block that the model provided, leading to execution errors. In a few cases, the execution silently fails with no output or errors. We concluded that the code is not executable, and we count these cases as failures.

5.1. Takeaways

In the following, we summarize some lessons learned and challenges faced during the replication study.

🔍 **Reliable Evaluation Tools.** We aimed to replicate the evaluation pipeline of the target paper as closely as possible, extending the provided codebase to support different models and infrastructures. Our goal was to focus on having a very reliable tool, allowing to re-run individual problem instances without restarting the entire process. We found it fundamental to have a reliable and well-tested evaluation tool, or at least, to have a starting basis to build upon. We encourage the research community to share their evaluation tools and pipelines to facilitate replication and comparison of results. A strategy could be to reuse and extend existing benchmarks such as LiveCodeBench [17], or consider adopting libraries like HuggingFace's *evaluate*[13].

🔍 **Fast and reliable inference infrastructure.** A significant amount of effort was devoted to setting up an inference infrastructure capable of running large models efficiently. We initially opted for *Ollama*, a popular and easy-to-use model serving tool. However, we found that its model compression format is not well suited for multi-GPU settings and complicates replication. We switched to *vLLM* to avoid provider-specific artifacts and ensure that our results could be replicated by others by using standard tools and the open-weights models offered by Hugging Face.

We deployed a single *vLLM* instance with tensor parallelism across our multi-GPU server, leading to a better load balancing and faster inference times compared to a single instance per GPU. Setting up such an environment is non-trivial and requires significant technical expertise. A useful contribution is

[13] https://github.com/huggingface/evaluate

to have a set of guidelines and scripts to facilitate the deployment and management of model inference infrastructures, such as having reference infrastructure-as-code artifacts.

🔍 **Rapid Model Evolution.** The exact versions of the models used in papers reporting empirical experiments were not always obtainable, some are no longer available or became unreasonably expensive (to force users to move to newer versions), and many have different knowledge cut-off dates due to continuous updates to their training data (*e.g.,* OpenAI models). Therefore, we find it important to prioritize open-weight models in evaluations, and specifically when the evaluation is not about the model, but about some pipeline on top of it. It is also important to document as much as possible the model version, knowledge cut-offs, and prompt settings to facilitate future replications. An effort in providing such a set of recommendations has been proposed by Baltes *et al.* [18].

Additionally, the model capabilities are becoming more and more advanced, and therefore some practices become obsolete. For example, using structured output with modern models leads to more reliable results rather than using YAML formatting. We initially implemented structured output parsing to mitigate formatting errors and parsing issues. This is true for modern models, such as *GPT-4o*, but led to inconsistent results with the models used in the experiment. To ensure a fair comparison with the original results, a good strategy for future work would be to choose a set of models having similar capabilities and cut-off date.

🔍 **Metric Limitations.** The standard metric for code generation tasks, `pass@k`, measures the ability to generate code that passes predefined test cases. However, it overlooks factors like the number and complexity of test cases, code quality, efficiency, and overall pipeline cost (*e.g.,* tokens used). This leads to an unfair comparison between simple baselines and more complex approaches like *AlphaCodium*. The same is true when there are differences in terms of capabilities of the model (*e.g., GPT-4* vs. *Codellama*), token usage (*AlphaCodium* uses four times more tokens), and inference time, which is crucial in practical applications.

We suggest considering additional metrics and qualitative analyses in these kinds of evaluations, such as weighting scores by test case count, measuring average time to solution, or assessing code quality and efficiency with static analysis tools. Considering problem complexity and test case coverage is also important, as simple test cases can inflate scores.

6. Threats to validity

In this section, we report the threats to the validity of our study.

Construct Validity. Even if we used most of the code and parameters from the original study, our modifications could have altered the final results. The obtained results are quantitatively consistent with those in the original paper. However, we cannot completely rule out since the outputs of LLMs can be highly variable even with the same inputs and parameters.

Internal Validity.

Although `pass@k` is widely used as a metric for evaluating code generation models [10], the sample selected in our experiment could have led to high variance in the results. However, a solid replication would have required far more computational resources and time. Since our goal was limited to exploring the replicability of the *AlphaCodium* approach, we acknowledged this threat when discussing the results.

Another threat concerns the choice of models. We selected models as close as possible to the originals (*i.e., Deepseek-Coder* and *GPT-3.5-Turbo*), adding *CodeLlama* for comparison. There is also a risk of knowledge overlap between model training data and the *CodeContests* dataset, potentially favoring some models. However, only *GPT-3.5-Turbo* should be unaffected by this, and its results are comparable to *Deepseek-Coder*, suggesting that the model has not been penalized.

External validity.

This threat mostly refers to the applicability of the discussed challenges and takeaways. Despite they are general suggestions applicable to LLM-based experiments, there could be cases in which some encountered problems are less prominent. For example, newer models have fewer issues in generating structured outputs.

7. Conclusion and Future Work

Test-Driven Development (TDD) has been shown to be an effective approach to enhance the performance of Large Language Models (LLMs) in solving competitive programming problems. In this paper, we presented a replication study of the work by Ridnik *et al.* [8], which introduced *AlphaCodium*, a TDD-based approach to improve LLMs' capabilities in this domain.

Our replication study extended the original implementation to support additional LLMs, including open-source models, and we evaluated their performance on the *CodeContests* dataset [5]. We obtained results that were generally consistent with those reported in the original paper and discussed the challenges and common errors encountered during the replication process.

However, in the process of building the software for the replication, and while running the replication itself, we also learned several details, which we have presented as observations and takeaways.

We plan to further investigate the applicability of different metrics for a broader and fair evaluation of LLM-based code generation approaches, as well as extend the evaluation to additional models and newer datasets. Last but not least, we also plan to release in the future the evaluation tool we are developing to help the community in replicating and benchmarking LLM-based code generation approaches.

8. Acknowledgements

The study presented in this paper was funded in part by the Advise project, funded by the Spanish AEI with reference 2024/00416/002.

References

[1] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, J. M. Zhang, Large language models for software engineering: Survey and open problems, in: 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE), IEEE, 2023, pp. 31–53.

[2] N. Nguyen, S. Nadi, An empirical evaluation of GitHub Copilot's code suggestions, in: Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 1–5.

[3] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, et al., Code Llama: Open foundation models for code, arXiv preprint arXiv:2308.12950 (2023).

[4] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, et al., DeepSeek-Coder: When the large language model meets programming–the rise of code intelligence, arXiv preprint arXiv:2401.14196 (2024).

[5] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, et al., Competition-level code generation with AlphaCode, Science 378 (2022) 1092–1097.

[6] S. Piya, A. Sullivan, LLM4TDD: best practices for test driven development using large language models, in: Proceedings of the 1st International Workshop on Large Language Models for Code, 2024, pp. 14–21.

[7] N. S. Mathews, M. Nagappan, Test-driven development and LLM-based code generation, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 1583–1594.

[8] T. Ridnik, D. Kredo, I. Friedman, Code generation with AlphaCodium: From prompt engineering to flow engineering, 2024. arXiv:2401.08500.

[9] CodiumAI, AlphaCodium: Official implementation for the paper "code generation with alphacodium", https://github.com/Codium-ai/AlphaCodium, 2024. Accessed: 2025-09-28.

[10] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., Evaluating large language models trained on code, arXiv preprint arXiv:2107.03374 (2021).

[11] AlphaCode Team, Google DeepMind, AlphaCode 2 technical report, https://storage.googleapis.com/deepmind-media/AlphaCode2/AlphaCode2_Tech_Report.pdf, 2023. Accessed: 2025-09-28.
[12] S. Piya, A. Samadi, A. Sullivan, Is more or less automation better? an investigation into the LLM4TDD process, in: 2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code), IEEE, 2025, pp. 161–168.
[13] S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, S. K. Lahiri, LLM-based test-driven interactive code generation: User study and empirical evaluation, IEEE Transactions on Software Engineering (2024).
[14] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al., Program synthesis with large language models, arXiv preprint arXiv:2108.07732 (2021).
[15] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, Y. Iwasawa, Large language models are zero-shot reasoners, Advances in neural information processing systems 35 (2022) 22199–22213.
[16] X. Gu, M. Chen, Y. Lin, Y. Hu, H. Zhang, C. Wan, Z. Wei, Y. Xu, J. Wang, On the effectiveness of large language models in domain-specific code generation, ACM Transactions on Software Engineering and Methodology 34 (2025) 1–22.
[17] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, I. Stoica, LiveCodeBench: Holistic and contamination free evaluation of large language models for code, arXiv preprint arXiv:2403.07974 (2024).
[18] S. Baltes, F. Angermeir, C. Arora, M. Muñoz Barón, C. Chen, L. Böhme, F. Calefato, N. Ernst, D. Falessi, B. Fitzgerald, et al., Guidelines for empirical studies in software engineering involving large language models, arXiv e-prints (2025) arXiv–2508.