



Original Software Publication



ICCheck: A Portable, Language-agnostic Tool for Synchronizing Code Clones

Motoki Abe , Shinpei Hayashi *

School of Computing, Institute of Science Tokyo, Ookayama 2-12-1, Meguro-ku, 152-8550, Tokyo, Japan

ARTICLE INFO

Keywords:

Code clone
Clone synchronization
Language Server Protocol

ABSTRACT

Inconsistent modifications to code clones can lead to software defects. Many approaches exist to support consistent modifications based on clone detection and/or change pattern extraction. However, no tool currently supports synchronization of code clones across diverse programming languages and development environments. We propose ICheck, a tool designed to be language-agnostic and portable across various environments. By leveraging an existing language-agnostic clone search technique and limiting the tool's external dependency to an existing Git repository, we developed a tool that can assist in synchronizing code clones in diverse environments. We validated the tool's functionality in multiple open-source repositories, where ICheck was able to detect overlooked clone modifications in over 30 programming and domain-specific languages and delivered interactive suggestions within a median of 0.27 seconds in editor environments, demonstrating its language independence and responsiveness. Furthermore, by supporting the Language Server Protocol, we confirmed that ICheck can be integrated into multiple development environments with minimal effort. ICheck is available at <https://github.com/salab/icheck>

Metadata

Nr.	Code metadata description	
C1	Current code version	v0.10.0
C2	Permanent link to code/repository used for this code version	https://github.com/salab/icheck/releases/tag/v0.10.0
C3	Permanent link to Reproducible Capsule	https://doi.org/10.5281/zenodo.15163079
C4	Legal Code License	MIT license
C5	Code versioning system used	Git
C6	Software code languages, tools, and services used	Go, Kotlin, TypeScript
C7	Compilation requirements, operating environments and dependencies	Go v1.23.0+
C8	If available, link to developer documentation/manual	https://github.com/salab/icheck/blob/main/README.md
C9	Support Email for questions	toki@se.comp.isct.ac.jp , hayashi@comp.isct.ac.jp

* Corresponding author.

E-mail address: hayashi@comp.isct.ac.jp (S. Hayashi).

1. Introduction

Code clones are identical or similar code fragments [1]. The presence of code clones is considered a code smell, and refactorings are often performed to eliminate them [2]. However, some code clones are not suitable for removal due to challenges in refactoring [3].

Cloned code fragments often require co-evolution. A set of cloned code fragments is called a clone set [4]. According to Krinke, in half of the cases where a modification is made to one fragment in a clone set, a consistent modification is also applied to all fragments in the set [5]. A typical case is that a code fragment with a defect is reused by copy-paste; in this case, the defect of the original fragment may remain in all its cloned fragments, and therefore the same fix may need to be applied to them as well [1]. Failing to consistently modify code clones in a clone set when co-evolution is required can lead to defects or inconsistent behavior [6]. To address this issue, various approaches have been proposed to support consistent synchronizations in a clone set, including clone tracking and change pattern extraction [7–10]. Here, *clone synchronization* refers to maintaining consistency of modifications across cloned fragments, which differs from *cloning*, the act of creating new duplicates.

However, these techniques and their tool implementations are often designed for specific development environments or programming languages (see the details in Section 2), so enormous effort is required to adapt these techniques to other environments or languages. Modern software development involves diverse programming languages [11] and development environments [12], while new languages, environments, and editors are consistently emerging. Existing approaches for supporting synchronizations cannot be readily applied in such diverse settings.

To address this issue, we propose ICCheck (Inconsistent Change Checker), a tool for assisting with code clone synchronizations that is highly portable and independent of both specific languages and development environments. In brief, ICCheck works by detecting changes in a Git repository, searching for cloned fragments similar to the modified code, and then identifying and reporting cases where a clone was not consistently updated. *The impact* of this software can be summarized as follows:

- *High portability.* ICCheck is applicable to any Git-managed repository that involves code text, regardless of the used programming languages. To ensure high portability while minimizing the loss of usability, it can function either as a well-structured command-line interface (CLI) or as a server compliant with the Language Server Protocol (LSP), offering interactive support during code editing. Preliminary experiments demonstrated ICCheck's efficiency and portability while maintaining its accuracy (Section 4). To our knowledge, no other tools currently support this combination of features. Beyond its strong practical utility, ICCheck is also suitable for empirical studies on clone synchronization across different programming languages.
- *Design choices to enable the portability* (Sections 2 and 3). To ensure the adaptability across a wide range of use cases and development environments, we carefully selected the underlying clone search algorithm, its usage patterns, and implementation strategies, explicitly documenting them as part of our design choices. We believe this provides valuable guidance for future implementations of clone synchronization techniques.

The remainder of this paper is structured as follows. Section 2 introduces a motivating example of code clones for designing ICCheck. Section 3 defines the requirements for ICCheck and presents its design and implementation. Section 4 preliminarily evaluates ICCheck. Section 5 concludes this paper.

2. Motivation

Code clones in various languages. Code clones exist not only in general-purpose programming languages such as Java and C but also in domain-specific languages like Dockerfile [13] and configuration files written in YAML or JSON. Fig. 1 illustrates an example of code clones found in YAML files¹. In this case, the file defines Kubernetes object definitions, and whenever modifications are made to a clone instance, all the other instances within the clone set must be updated consistently. As this example demonstrates, it is not uncommon for code clones to exist across multiple files, even in domain-specific languages with unique syntaxes or in configuration files like YAML and JSON. As prior studies have shown, code clones exist in a wide range of programming languages [14, 15], highlighting the need for clone management and synchronization techniques applicable across diverse development contexts. Moreover, consistent modifications are sometimes required for these clones as well. Therefore, it is essential to support the detection and consistent modification of code clones not only in mainstream programming languages but also in domain-specific languages and data serialization formats used for configurations. Note that although there is also a demand for detecting clones that span across multiple languages [16–18], our tool does not aim to address the synchronization of such cross-language clones.

Diversity of use cases and development environments. In modern software development, code changes are reviewed in various scenarios, including confirming them while editing in an editor, reviewing a sequence of commits in a branch before submitting a pull request, and analyzing them on Continuous Integration/Delivery (CI/CD) platforms. Developers use a variety of tools for editing and validating source code, ranging from simpler code editors to advanced text editors and integrated development environments (IDEs) such as Visual Studio Code (VSCode) and IntelliJ IDEA. Additionally, numerous CI/CD platforms, such as GitHub Actions, are used for validating and revising source code. To effectively support simultaneous modifications to code clones in real-world software development, a highly portable tool is required: one that integrates seamlessly with various text editors and can also be executed in CI/CD environments.

¹ <https://github.com/motoki317/manifest/tree/6e13a913fb8269e0d80d9a6733c3eab7bbcc7cfa>

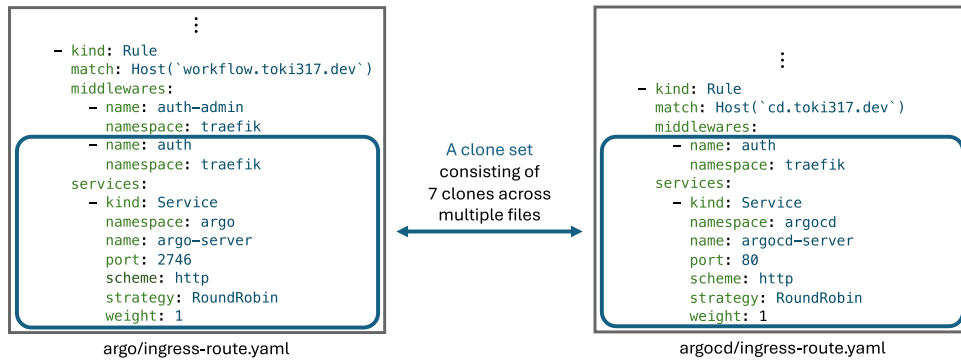


Fig. 1. Clone set in YAML files.

Table 1

Comparison of approaches for supporting clone synchronization.

Name	Languages	Environments	Tool availability
Simultaneous Editing [19]	Java, HTML	LAPIS	Available (https://groups.csail.mit.edu/graphics/lapis/)
Linked Editing [7]	Java	XEmacs	Not specified
CloneTracker [8,20]	Java	Eclipse	Available (https://www.cs.mcgill.ca/~swevo/clonetracker/)
CReN [21]	Java	Eclipse	Link broken
JSync [22]	Java	Eclipse	Not specified
CCSync [23]	Java	Not specified	Not specified
CCDemon [24]	Java	Eclipse	Available (https://github.com/llmhy/ccdemon)
Clone Notifier [25]	Covered by 3 detectors	Custom GUI	Available (https://github.com/s-tokui/CloneNotifier)
CLIONE [26]	Java, Kotlin, Python, C++	CI (GitHub App)	Available (https://github.com/T45K/CLIONE)
CloneTracker (Commercial) [27]	10 languages	Custom GUI	Available (https://clonetracker.com/en/)
ICCheck (our tool)	Language-agnostic	CLI, CI, LSP clients	Available (https://github.com/salab/iccheck)

Based on these motivations, the following key aspects are crucial for supporting consistent modifications of code clones in practical software development: 1) the support for simultaneous modification of code clones in minor domain-specific languages and configuration file formats, and 2) high portability, enabling usage across diverse use cases and development environments.

Despite the advancements in code clone research and tools, to the best of our knowledge, no existing tool meets these requirements. Table 1 provides a comparison of existing approaches for supporting simultaneous modifications based on code clones, including those covered in a recent survey of clone tracking techniques [28], revealing the supported languages and environments, and the tool availability. While various techniques and tools have been proposed, most of them are limited to Java and are implemented for specific IDEs, such as Eclipse or custom editors. Furthermore, some tools are not publicly available.

3. ICCheck in a nutshell

3.1. Design

The key design choices in the development of ICCheck are outlined below.

(1) Integrating language-agnostic clone search technique. ICCheck detects source code changes made by developers, identifies code clones containing similar code snippets with the changed ones, and notifies the developer of their presence. For this purpose, clone search, where a set of modified code fragments serves as a query to find similar code fragments, is more suitable than full-scale clone detection, which identifies duplicated sections across an entire codebase. However, a language-agnostic clone search method is required to ensure adaptability to any programming language without additional effort. In particular, detecting *micro-clones*, which are code clones shorter than five lines of code, is crucial [3]. To achieve this, ICCheck employs FLeCCS [29], a language-independent technique that identifies clones based on textual similarity with the given query code fragment. FLeCCS compares all files in a target project using a sliding window approach. It calculates bigram sets at the character level for each line in the query and the target code snippet, then it determines similarity using the weighted average of Dice coefficients with weighting based on line length. By default, a pair of code fragments with a similarity score of 0.7 or higher is regarded as a code clone, following the default setting of FLeCCS; however, this threshold can be changed through a user-specified option.

(2) Adherence to Git-based code change processes. To efficiently apply ICCheck across various code change scenarios while keeping implementation costs low, it leverages Git, a widely used version control system, to track changes. The relevant use cases discussed in Section 2 can be recognized as specific changes within the version control context. By integrating with Git, ICCheck seamlessly incorporates the support of clone synchronization into the most common software development workflows today. ICCheck identifies a changeset by specifying commit IDs, branch names, or the working tree before committing. It searches for related code clones within the project using the changeset as a query and notifies the developer of their presence. In addition, ICCheck operates solely on a Git repository, without requiring any precomputed clone database or additional intermediate artifacts. By ensuring that it

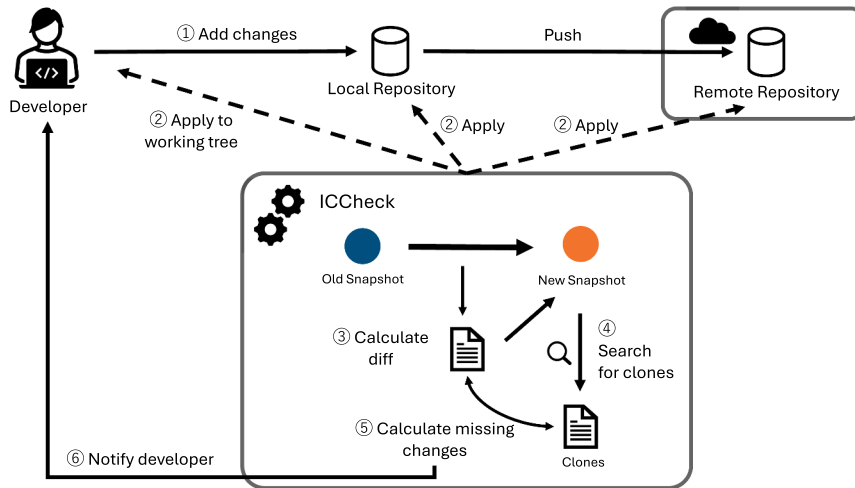


Fig. 2. Workflow of ICCheck.

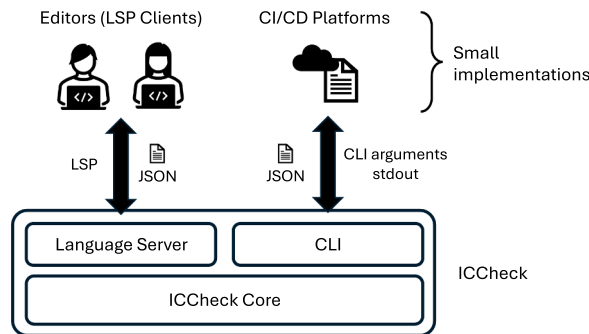


Fig. 3. Architecture of ICCheck.

performs cleanly without leaving any analysis artifacts, ICCheck remains simple to use and easy to integrate into existing workflows, thereby reducing the barrier to adoption. We therefore intentionally chose not to adopt more computationally intensive, history-aware clone detection approaches that verify the co-evolution of cloned fragments across versions, which are typically heavy and expensive as they rely on precomputed clone databases or repeated cross-version analyses.

(3) Providing CLI and Language Server. To interactively support clone synchronization in diverse environments, ICCheck offers both a Command Line Interface (CLI) and integration with the Language Server Protocol (LSP) [30], which is supported by many modern development environments. Clone search can be executed via the CLI by specifying a Git repository and changeset, and the results can be in human-readable text or machine-friendly JSON formats. By leveraging CLI output and exit codes, ICCheck can be integrated with CI/CD workflows such as GitHub Actions and commit-hooks. Additionally, by implementing a Language Server, ICCheck enables integration with multiple editors, including VSCode and IntelliJ, allowing interactive support during code editing to detect potential overlooked changes with reasonable implementation cost.

The workflow of ICCheck is summarized in Fig. 2. The developer adds changes to source code in a Git repository (①) and runs ICCheck, which conceptually takes two snapshots as input to define the change range (②). Then, ICCheck computes code differences between the two snapshots (③). Using the modified code as a query, ICCheck searches for code clones in the new snapshot (④). It detects overlooked change opportunities by comparing the identified clones with the modified code (⑤). Any unchanged clone instances are considered overlooked, and they are notified to the developer (⑥).

3.2. Implementation

To distribute both the CLI and Language Server as a single executable file while supporting multiple operating systems (Windows, macOS, and Linux) and architectures (amd64 and arm64), ICCheck was implemented from scratch using Golang. The underlying clone search algorithm, FLeCCS, was reimplemented in Golang. Although it was originally implemented as a Java GUI application, we did not use it for the ease of implementing the integration with Git. The latest version (v0.9.0 at the time of writing) is available on the GitHub releases page with its executables.

The architecture of ICCheck is illustrated in Fig. 3. The CLI and Language Server modules handle external input and output. The Core module performs clone search and computes overlooked changes. This modular design keeps the implementation simple and

```

$ iccheck
2024/11/26 22:43:13 INFO 22 change chunk(s) within 3 file(s) found.
    from="HEAD (d46bf7e87fb62877f0052534659589ecc4c8aa41)" to=WORKTREE
2024/11/26 22:43:13 INFO 5 clone(s) are likely missing consistent change.

Clone set #0 - 5 out of 6 clones are likely missing consistent change(s).
Missing changes (5):
  pkg/lsp/handler.go:74 (L74-L74)
  pkg/lsp/handler.go:93 (L93-L93)
  pkg/lsp/handler.go:112 (L112-L112)
  pkg/lsp/handler.go:147 (L147-L147)
  pkg/lsp/handler.go:52 (L52-L52)
Changed clones (1):
  pkg/lsp/handler.go:167 (L167-L167)

```

Fig. 4. Usage of ICCheck CLI.

maintainable. By providing a Language Server, ICCheck enables integration with LSP clients. Additionally, the CLI module's output can be structured in JSON format, making it easy to process programmatically. This simplifies CI/CD setup, allowing users to format the output for GitHub Actions with simple one-line shell commands.

ICCheck aims to provide suggestions within 1 second in an editor environment [31] and within 1 minute via the CLI. To achieve this, several optimizations were implemented such as parallel clone search using goroutines. When running as a Language Server, incremental clone search for code differences is performed, and ICCheck sleeps if the execution time exceeds a threshold and resumes when the developer's keystrokes pause to mitigate the runtime load in the developer's local environment.

3.2.1. The CLI

ICCheck is distributed as a single executable file with basic usage instructions documented in its README in GitHub. For example, when the CLI is triggered at a Git-managed directory with a modified working tree, it outputs related code clones and missing changes as shown in Fig. 4. In addition to the basic workflow described in Section 3.1, the tool first checks whether the current directory is Git-managed and, if so, sets the detected repository as the search target. The target repository can also be changed using a CLI option. Although the before- and after-change snapshots can be explicitly specified via CLI options, if they are not provided, the tool automatically determines the most appropriate comparison based on the state of the Git repository. In this example, since the working tree was modified but not yet committed, the comparison was made between the HEAD commit and the working tree. Finally, the identified suspicious clones are displayed as “missing changes”.

Executing `iccheck --help` provides a list of CLI options and their descriptions. The `--repo`, `--from`, and `--to` options specify the Git repository and the snapshots before and after changes, respectively. If `--from` is not specified, ICCheck automatically uses the parent of the commit specified by `--to` as the snapshot before changes. The `--ignore` option allows filtering of the reported missing changes, whereas the `--include` option limits the paths to be analyzed. The `--format` option specifies the output format with `--format json` being particularly useful for CI/CD integration.

Output clone instances are displayed in the order they are detected, i.e., according to the order of files processed, since ICCheck does not employ a specific ranking algorithm. Users who prefer a particular ordering can still implement their own ranking criteria, such as those proposed by Hamid et al. [32], as a post-processor of the JSON output.

3.2.2. The language server

ICCheck integrates with any LSP-compatible editor via its Language Server. Some LSP-compatible editors can utilize ICCheck without requiring special plugins simply by configuring the Language Server binary path. For environments that this simple configuration is not available, we implemented simple adapter plugins for VSCode² and IntelliJ IDEA³ to make ICCheck runnable easily.

When changes are made to a file within a Git repository, the Language Server detects the changes and searches for corresponding code clones. For example, in the YAML file shown in Fig. 5(a), modifying the `port` field on Line 18 triggers a warning suggesting a change to the unchanged micro-clone at Line 31. Additionally, as illustrated in Fig. 5(b), hovering the cursor over a warning and using the “Find References” feature allows quick navigation to all corresponding clone locations, without leaving the code editor. These warnings can serve as mild, real-time hints rather than strict alarms. In this sense, they help developers remain aware of potentially relevant clones while coding, without interrupting their workflow or requiring exhaustive inspection of all reported fragments.

3.2.3. Implementation strategies and optimizations

To improve runtime performance and scalability, ICCheck incorporates several design and implementation optimizations, as summarized below.

- During detection, temporary data used for searching *within each file* are discarded immediately after that file's search completes. As a result, the only data accumulated in memory across the execution are the detected missing changes to be reported, enabling

² <https://marketplace.visualstudio.com/items?itemName=motoki317.iccheck>

³ <https://plugins.jetbrains.com/plugin/24779-iccheck--inconsistency-check>

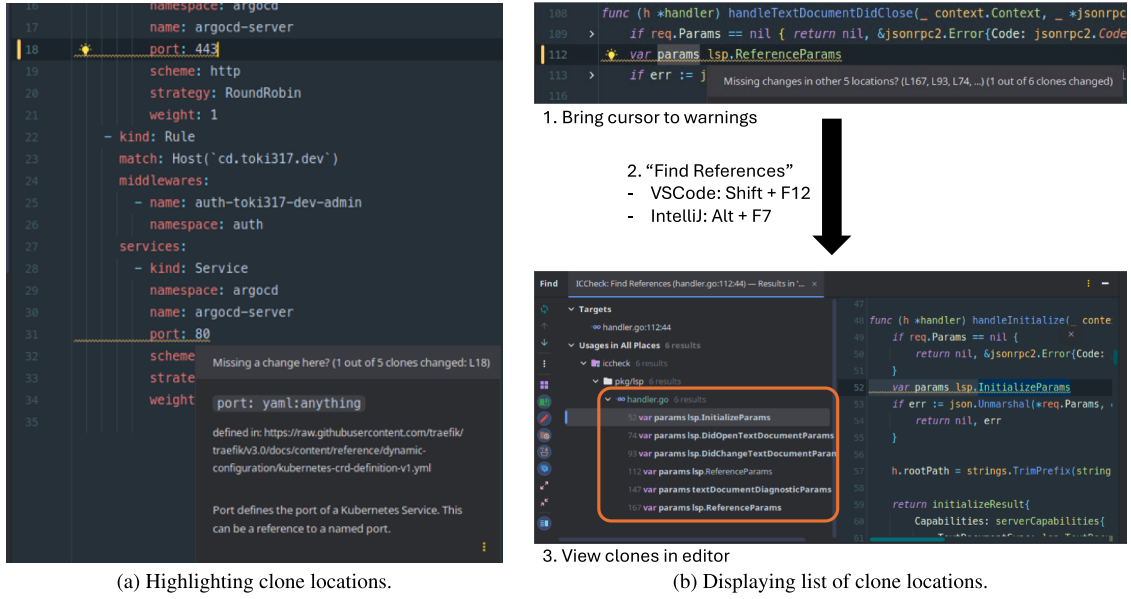


Fig. 5. Supporting clone synchronization via ICCheck Language Server.

Table 2
Releases of ICCheck.

Release Date	Version	Summary
2024-05-29	v0.1.0	Initial release
2024-06-20	v0.3.0	LSP functionality
2024-10-21	v0.5.0	LSP performance improvements
2024-11-17	v0.6.0	Displaying clone locations in LSP
2024-11-26	v0.7.0	Suggestion exclusion feature, CLI option reorganization
2024-12-05	v0.7.1-0.7.6	Performance optimizations
2025-01-12	v0.8.0-0.8.2	Performance optimizations and interface improvements
2025-01-26	v0.9.0	CI workflow and minor enhancements
2025-10-14	v0.10.0 (Latest)	CLI option for specifying the path to be included

detection without sustained memory pressure. For example, when applied to the latest commit of the Linux kernel repository (41.3M lines of code)⁴, the analysis completed using only 3.7 GB of memory.

- We have ensured that the implementation of the hot path in the FLeCCS algorithm is efficient enough. In particular, the bigram set of each line is represented as a sorted integer array, and intersections are counted via a two-pointer scan, which significantly reduces computational overhead compared with set-based approaches.
- We extended the *go-git* library in Go to parallelize the loading of Git tree structures, accelerating traversal and metadata reads on large repositories.
- In LSP mode, detection can be triggered on every edit. To avoid redundant work, per-file results from past queries are cached and reused when a file remains unchanged under the current edit sequence; only invalidated files are recomputed. Additionally, when ICCheck runs longer, the process inserts short sleeps to avoid monopolizing CPU and to prevent disruption of developer activities.

3.3. Evolution of ICCheck

Since its initial release (v0.1.0), ICCheck has undergone continuous development for approximately eight months. A total of 34 releases have been made, focusing on performance improvements, bug fixes, and feature enhancements, contributing to its maturity as a tool. Table 2 summarizes the key releases in the evolution of ICCheck. For example, the suggestion exclusion setting was introduced in v0.7.0. Previous research [33] has shown that not all fragments in a clone set always need to be evolved for the same reason. Through our usage experiences, we found that automatically generated files and `import` statements were frequently reported as false positives. To address this issue, a feature allowing exclusions via CLI options and/or configuration files was implemented.

Furthermore, improvements were made based on over two months of practical use by multiple practitioners and the feedback obtained. They provided positive feedback on ICCheck's ability to identify missing changes in YAML configurations and eRuby

⁴ <https://github.com/torvalds/linux/commit/9b332ce>

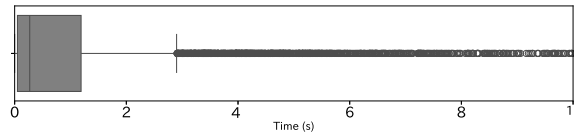


Fig. 6. Execution time of ICCheck (seconds, $n = 9314$).

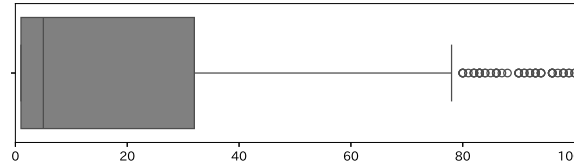


Fig. 7. Number of detected missing changes ($n = 2677$).

templates during new feature development involving copy-pasting. The capability to extend beyond standard programming languages makes it particularly useful for maintaining consistency in diverse file types.

4. Preliminary evaluation

We quantitatively evaluated whether ICCheck meets the required criteria from three perspectives: execution time, portability to various languages, and accuracy. In the following experiments, we used ICCheck v0.8.2, which was the latest version available at the time of the evaluation⁵, on Ubuntu 24.10 (WSL2) running on an i9-12900K (16 cores, 24 threads) with 16 GB RAM. Several scripts for reproducing the experimental results are provided in the repository⁶.

4.1. Execution time

Based on the GitHub Ranking as of August 7, 2024 [34], we selected the top three repositories from each of the 34 language categories, totaling 102 repositories. Although this star-based selection approach is not highly reliable and may include repositories that are not necessarily well-engineered projects [35], we prioritized its simplicity and ease of obtaining key data under the constraint of selecting repositories across a wide variety of programming languages. For each repository, we analyzed the latest 100 commits on the default branch. However, commits with more than 25 modified files were excluded, resulting in a total of 9368 commits being analyzed for missing changes. This threshold of commit size was heuristically chosen to exclude large-scale commits, which are likely to involve perfective maintenance operations [36], such as refactorings or search-and-replace edits, rather than ordinary interactive coding activities, as such large commits may introduce noise into the analysis. The timeout for ICCheck was set to 60 seconds.

Among the 9368 commits, 9314 (99.4%) completed the detection without timing out. The distribution of their execution times are shown in Fig. 6. For easier viewing, the x-axis is cropped to 10 seconds. The median execution time was 0.27 seconds with an average of 2.12 seconds. 75% of the cases completed within 2 seconds. However, in some cases with large changesets or substantial repository sizes, detection took longer. For example, the detection in `torvalds/linux` took at least 14 seconds per commit.

In conclusion, *ICCheck CLI can provide suggestions with a practical response time for most repositories.*

4.2. Portability to various languages

Analyzing the results from the 9314 commits that did not time out, missing changes were detected in 2677 commits (28.7%). As shown in Fig. 7, the median number of detected missing changes per commit was 5, the average was 225, the maximum was 53,005, and the total was 602,416. Due to the large maximum value, the axis in the figure is cropped at 100. As shown in Fig. 8, change suggestions were confirmed for all 34 programming languages used in the study. Note that these repositories also included data formats such as JSON, which were not included in the selected 34 languages, and suggestions for them were also observed.

In conclusion, *ICCheck is capable of assisting with clone synchronization across multiple languages.*

4.3. Accuracy for open-source repositories

From the suggestions obtained in the previous evaluation on the portability, we randomly sampled 100 clone sets and manually validated them. As a result, 63 out of 100 were correct code clones, of which 49 required consistent synchronization for all their clone

⁵ The updates from the version evaluated in this study (v0.8.2) to the latest version (v0.10.0) include a minor improvement to the detection behavior that may affect the results of the experiments on open-source repositories (Sections 4.1, 4.2, and 4.3), although the impact is expected to be small and unlikely to affect the conclusions in these subsections.

⁶ <https://github.com/salab/iccheck/tree/main/docs/evaluations>

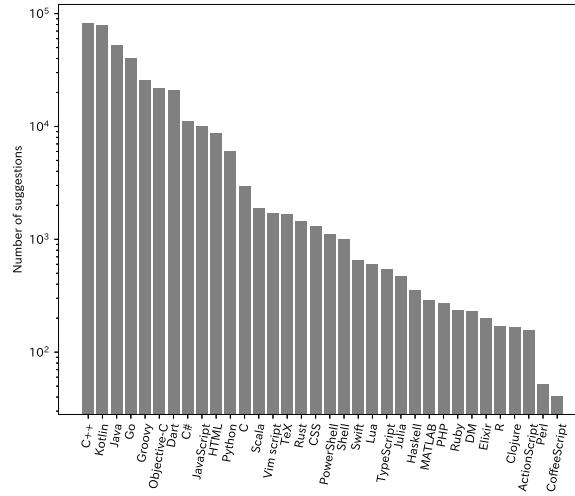


Fig. 8. Number of suggestions per programming language.

instances, and 14 were not. Meanwhile, 37 were non-clones, i.e., false positives, consisting of unrelated similar code or similar lines formed by auto-generated files, resulting in incorrect suggestions. The correct clone set suggestions included languages such as TeX, CSS, JavaScript, TypeScript, Dart, Vim script, PowerShell, Python, Julia, Lua, and Go. Moreover, cross-language change suggestions were also confirmed, such as when code snippets in Markdown triggered suggestions for TeX snippets.

In conclusion, *ICCheck results included meaningful clone suggestions in almost half of the cases, covering multiple languages.*

4.4. Accuracy for the CBCD dataset

We used the dataset of the CBCD approach [37], which involves 53 C language clone sets containing bugs. Note that we also referred to the NCDSearch dataset [38] for the location information of the bugs in the CBCD dataset. Out of these 53 cases, we excluded 15 clone sets that did not exist within the same snapshot and evaluated the remaining 38 cases. The average precision and recall per bug were 0.281 and 0.289, respectively. While there are 69 correct bug locations to predict across the entire dataset, ICCheck produced 23 suggestions, 16 of which were correct. Thus, the suggestion-based precision was 0.696, and recall was 0.232. The baseline approach, FLeCCS [29], reports a precision and recall of around 0.6 and 0.5, respectively. Although the obtained precision, including that of the baseline approach, is not particularly high, this may partly stem from our design choice of a lightweight, on-demand approach that analyzes a single snapshot rather than maintaining historical clone data across versions. Despite this trade-off, ICCheck's suggestion mechanism provides practically useful interactive assistance by identifying potentially overlooked clone modifications in practical development scenarios.

In conclusion, although the evaluation targets differ, *the precision of ICCheck was comparable to the evaluation results of FLeCCS.*

5. Conclusion

By utilizing a language-independent clone search technique, limiting external dependencies to Git, and adopting a CLI-based input/output approach, we designed and prototyped ICCheck, a highly portable tool for supporting code clone synchronization. Furthermore, by incorporating Language Server functionality, we demonstrated that ICCheck can be easily integrated with various editors. We also provided a quantitative evaluation showing that ICCheck can support clone synchronization in multiple languages within a practical execution time and that its suggestion accuracy is reasonable.

As revealed in the execution time evaluation, execution time increases in proportion to the volume of code differences. This is because ICCheck treats each fragment of a code difference as a query and performs clone search across the entire snapshot referenced by the target commit. Optimizing the suggestion algorithm may help improve execution time. Another direction for improvement is to conduct a sensitivity analysis to empirically identify a more appropriate default similarity threshold that contributes to overall accuracy, as well as a comprehensive comparative evaluation of ICCheck against existing clone-synchronization approaches to assess their relative effectiveness and efficiency in detecting overlooked clone modifications across diverse programming languages. Other future work includes improving the usability and stability of ICCheck by continuing and expanding its use in practical environments, which will help us fix bugs and add new features. After these improvements, conducting more formal user studies and surveys to systematically evaluate the practical usefulness and usability of ICCheck in real development environments will be valuable.

Declaration of generative AI and AI-assisted technologies in the writing process

During the preparation of this work the authors used ChatGPT in order to improve readability and language of the work. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

CRedit authorship contribution statement

Motoki Abe: Writing – original draft, Validation, Software, Resources, Methodology, Investigation, Formal analysis, Conceptualization; **Shinpei Hayashi:** Writing – original draft, Supervision, Project administration, Methodology, Funding acquisition, Conceptualization.

Acknowledgments

This work was partly supported by JSPS KAKENHI (JP23K24823, JP25K03102, JP25H01125, JP24H00692, and JP21KK0179). We would like to thank Dr. Michael J. Decker for his comments on the earlier version of this manuscript.

References

- [1] C.K. Roy, J.R. Cordy, A survey on software clone detection research, Technical Report 2007-541, School of Computing, Queen's University, 2007. <https://research.cs.queensu.ca/TechReports/Reports/2007-541.pdf>.
- [2] M. Fowler, Refactoring: Improving the design of existing code, Addison-Wesley Professional, 2nd edition, 2018.
- [3] M. Mondai, C.K. Roy, K.A. Schneider, Micro-clones in evolving software, in: Proc. 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'18), 2018, pp. 50–60.
- [4] K. Inoue, C.K. Roy (Eds.), Code Clone Analysis Research, Tools, and Practices: Research, Tools, and Practices, Springer Singapore, 2021.
- [5] J. Krinke, A study of consistent and inconsistent changes to code clones, in: Proc. 14th Working Conference on Reverse Engineering (WCRE'07), 2007, pp. 170–178.
- [6] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, Do code clones matter?, in: Proc. 31st International Conference on Software Engineering (ICSE'09), 2009, pp. 485–495.
- [7] M. Toomim, A. Begel, S.L. Graham, Managing duplicated code with linked editing, in: Proc. IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC'04), 2004, pp. 173–180.
- [8] E. Duala-Ekoko, M.P. Robillard, Tracking code clones in evolving software, in: Proc. 29th International Conference on Software Engineering (ICSE'07), 2007, pp. 158–167.
- [9] N. Meng, M. Kim, K.S. McKinley, Sydit: Creating and applying a program transformation from an example, in: Proc. 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11), 2011, pp. 440–443.
- [10] Y. Ueda, T. Ishio, K. Matsumoto, DevReplay: Linter that generates regular expressions for repeating code changes, *Sci. Comput. Program.* 223 (2022) 102857:1–10.
- [11] S. Cass, The top programming languages 2024: Typescript and Rust are among the rising stars, 2024, (IEEE Spectrum, <https://spectrum.ieee.org/top-programming-languages-2024>).
- [12] M. Wessel, T. Mens, A. Decan, P.R. Mazrae, The GitHub development workflow automation ecosystems, in: Software Ecosystems: Tooling and Analytics, Springer Cham, 2023, pp. 183–214.
- [13] T. Tsuru, T. Nakagawa, S. Matsumoto, Y. Higo, S. Kusumoto, Type-2 code clone detection for Dockerfiles, in: Proc. 15th IEEE International Workshop on Software Clones (IWSC'21), 2021, pp. 1–7.
- [14] Y. Semura, N. Yoshida, E. Choi, K. Inoue, CCFinderSW: Clone detection tool with flexible multilingual tokenization, in: Proc. 24th Asia-Pacific Software Engineering Conference (APSEC'17), 2017, pp. 654–659.
- [15] W. Zhu, N. Yoshida, T. Kamiya, E. Choi, H. Takada, Development and benchmarking of multilingual code clone detector, *J. Syst. Softw.* 219 (2025) 112215:1–20.
- [16] D. Perez, S. Chiba, Cross-language clone detection by learning over abstract syntax trees, in: Proc. 16th IEEE/ACM International Conference on Mining Software Repositories (MSR'19), 2019, pp. 518–528.
- [17] K.W. Nafi, T.S. Kar, B. Roy, C.K. Roy, K.A. Schneider, CLCDSA: Cross language code clone detection using syntactical features and API documentation, in: Proc. 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19), 2019, pp. 1026–1037.
- [18] T. Vislavski, G. Rakić, N. Cardozo, Z. Budimac, LICCA: A tool for cross-language clone detection, in: Proc. 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'18), 2018, pp. 512–516.
- [19] R.C. Miller, B.A. Myers, Interactive simultaneous editing of multiple text regions, in: Proc. USENIX Annual Technical Conference, 2001, pp. 161–174.
- [20] E. Duala-Ekoko, M.P. Robillard, CloneTracker: Tool support for code clone management, in: Proc. 30th International Conference on Software Engineering (ICSE'08), 2008, pp. 843–846.
- [21] P. Jablonski, D. Hou, CReN: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE, in: Proc. OOPSLA Workshop on Eclipse Technology EXchange (ETX'07), 2007, pp. 16–20.
- [22] H.A. Nguyen, T.T. Nguyen, N.H. Pham, J. Al-Kofahi, T.N. Nguyen, Clone management for evolving software, *IEEE Trans. Softw. Eng.* 38 (5) (2012) 1008–1026.
- [23] X. Cheng, H. Zhong, Y. Chen, Z. Hu, J. Zhao, Rule-directed code clone synchronization, in: Proc. 24th IEEE International Conference on Program Comprehension (ICPC'16), 2016, pp. 1–10.
- [24] Y. Lin, X. Peng, Z. Xing, D. Zheng, W. Zhao, Clone-based and interactive recommendation for modifying pasted code, in: Proc. 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15), 2015, pp. 520–531.
- [25] S. Tokui, N. Yoshida, E. Choi, K. Inoue, Clone Notifier: Developing and improving the system to notify changes of code clones, in: Proc. 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'20), 2020, pp. 642–646.
- [26] T. Nakagawa, Y. Higo, S. Kusumoto, CLIONE: Clone modification support for pull request based development, in: Proc. 27th Asia-Pacific Software Engineering Conference (APSEC'20), 2020, pp. 455–459.
- [27] S. Miki, H. Otsu, A. Asahara, T. Osawa, S. Chiba, Toward a code clone detection/tracking system used in industry (in Japanese), in: Proc. 40th JSSST Annual Conference, 2023. <https://jssst.or.jp/files/user/taikai/2023/papers/32-R.pdf>.
- [28] M. Mondal, C.K. Roy, K.A. Schneider, A survey on clone refactoring and tracking, *J. Syst. Softw.* 159 (2020) 110429:1–27.
- [29] M. Mondal, C.K. Roy, B. Roy, K.A. Schneider, FLeCCS: A technique for suggesting fragment-level similar co-change candidates, in: Proc. 29th IEEE/ACM International Conference on Program Comprehension (ICPC'21), 2021, pp. 160–171.
- [30] Microsoft, Official page for Language Server Protocol, (<https://microsoft.github.io/language-server-protocol/>).
- [31] S.K. Card, G.G. Robertson, J.D. Mackinlay, The information visualizer, an information workspace, in: Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI'91), 1991, pp. 181–186.

- [32] A.A. Hamid, M.F. Haque, M. Mondal, Ranking co-change candidates suggested by FLeCCS using programmer sensitivity, *Sci. Comput. Program.* 240 (2025) 103216:1–21.
- [33] F. Zhang, S.-C. Khoo, X. Su, Predicting consistent clone change, in: *Proc. 27th IEEE International Symposium on Software Reliability Engineering (ISSRE'16)*, 2016, pp. 353–364.
- [34] E. Li, *Github Ranking: Github stars and forks ranking list*, (<https://github.com/EvanLi/Github-Ranking>).
- [35] N. Munaiah, S. Kroh, C. Cabrey, M. Nagappan, Curating GitHub for engineered software projects, *Empiric. Softw. Eng.* 22 (6) (2017) 3219–3253.
- [36] A. Hindle, D.M. German, R. Holt, What do large commits tell us? A taxonomical study of large commits, in: *Proc. 5th International Working Conference on Mining Software Repositories (MSR'08)*, 2008, pp. 99–108.
- [37] J. Li, M.D. Ernst, CBCD: Cloned buggy code detector, in: *Proc. 34th International Conference on Software Engineering (ICSE'12)*, 2012, pp. 310–320.
- [38] T. Ishio, N. Maeda, K. Shibuya, K. Iwamoto, K. Inoue, NCDSearch: Sliding window-based code clone search using Lempel-Ziv Jaccard distance, *IEICE Trans. Inf. Syst.* E105–D (5) (2022) 973–981.