PyClone-Stream: An Automated and Self-growing Python Semantic Clone Dataset

Nigar ALIZADA †, Shiyu YANG†, and Yoshiki HIGO†

† Graduate School of Information Science and Technology, The University of Osaka, 1–5, Yamadaoka, Suita-shi, 565–0871 Japan

E-mail: †{nigar,yangsy,higo}@ist.osaka-u.ac.jp

Abstract Semantic code clone detection (Type-4 clones) remains one of the most challenging problems in clone detection research. Despite significant advances in machine learning and neural code representation models, progress is limited by the scarcity of large-scale, high-quality datasets—particularly for dynamically typed languages such as Python. Existing clone datasets are typically static, manually curated, small in scale, or predominantly composed of syntactic clones. This paper presents PyClone-Stream, an automated and continuously growing dataset of Python semantic code clones constructed from competitive programming submissions. PyClone-Stream leverages daily scraping of the AtCoder platform, persistent metadata storage, filesystem-based code archiving, and a multi-stage clone filtering pipeline. Syntactic clones (Type-1–3) are systematically removed using the NiCad clone detection tool, while the remaining clone pairs are labeled as candidate Type-4 clones. By supporting continuous dataset growth, PyClone-Stream enables long-term analysis and provides a scalable foundation for training and evaluating learning-based semantic clone detection models.

Key words semantic code clones, dataset, python

1. Introduction

Code cloning is prevalent in software systems, and its formation may stem from intentional code copying or from developers independently implementing the same functionality without realizing that an equivalent implementation already exists. Although some degree of cloning is intentional or unavoidable, unmanaged code clones increase maintenance costs, reduce readability, and facilitate the propagation of defects [1].

Code clones form a spectrum from syntactically identical Type-1 clones to semantically equivalent but syntactically diverse Type-4 clones. Accordingly, they are categorized into four types based on the degree of syntactic and semantic similarity [2]. While Type-1 to Type-3 clones can typically be detected using text-, token-, or tree-based techniques, Type-4 clones (also referred to as semantic clones) require reasoning about program semantics because they implement equivalent functionality while differing substantially in syntax and structure.

The absence of syntactic markers makes the detection of Type-4 clones particularly challenging and renders traditional clone detection techniques ineffective. Although recent learning-based approaches and pre-trained models such as CodeBERT, GraphCodeBERT, and CodeT5 [3]–[5] have demonstrated promising semantic modeling capabilities, their effectiveness is fundamentally constrained by the availability of large-scale, high-quality labeled datasets. In practice, **data availability has emerged as the primary bottleneck** limiting progress in semantic clone detection research.

Existing code clone datasets are often static, manually curated, limited in scale, or dominated by syntactic clone pairs [7].These limitations are particularly pronounced for dynamically typed languages such as Python, where flexible typing, rich standard libraries, and diverse implementation styles further complicate semantic equivalence. Despite its prevalence in real-world software systems, Python remains comparatively underexplored in existing semantic code clone datasets, motivating the selection of Python as the focus of this work. However, creating a robust benchmark requires careful consideration of data quality; prior studies have shown that improper dataset usage and labeling assumptions can significantly distort conclusions in semantic clone detection research [6].

This paper presents *PyClone-Stream* as a response to this gap: an automated and continuously growing dataset construction framework designed to support large-scale analysis and learning-based detection of Python semantic code clones.

2. Problem Definition

The main purpose of semantic code clone detection, commonly known as Type-4 clone detection, is to identify code snippets that implement the same functionality despite syntactic and structural differences. Progress in this area is constrained not by modeling capability alone, but by limitations in available datasets. Existing semantic clone datasets typically suffer from three fundamental shortcomings:

(i) they are static snapshots that do not reflect evolving code distributions [7], [10],

(ii) they rely heavily on manual annotation [10] or synthetic generation [11], limiting scalability, and

(iii) they insufficiently capture semantic diversity in dynamically typed languages such as Python [7].

These limitations are particularly problematic for learning-based approaches, which require large, diverse, and up-to-date training data to generalize effectively. The difficulty in achieving such generalization stems from a core and underexplored challenge: the continuous evolution of software ecosystems, which renders fixed benchmarks increasingly unrepresentative. New versions of the programming languages, programming problems, libraries, coding idioms, and

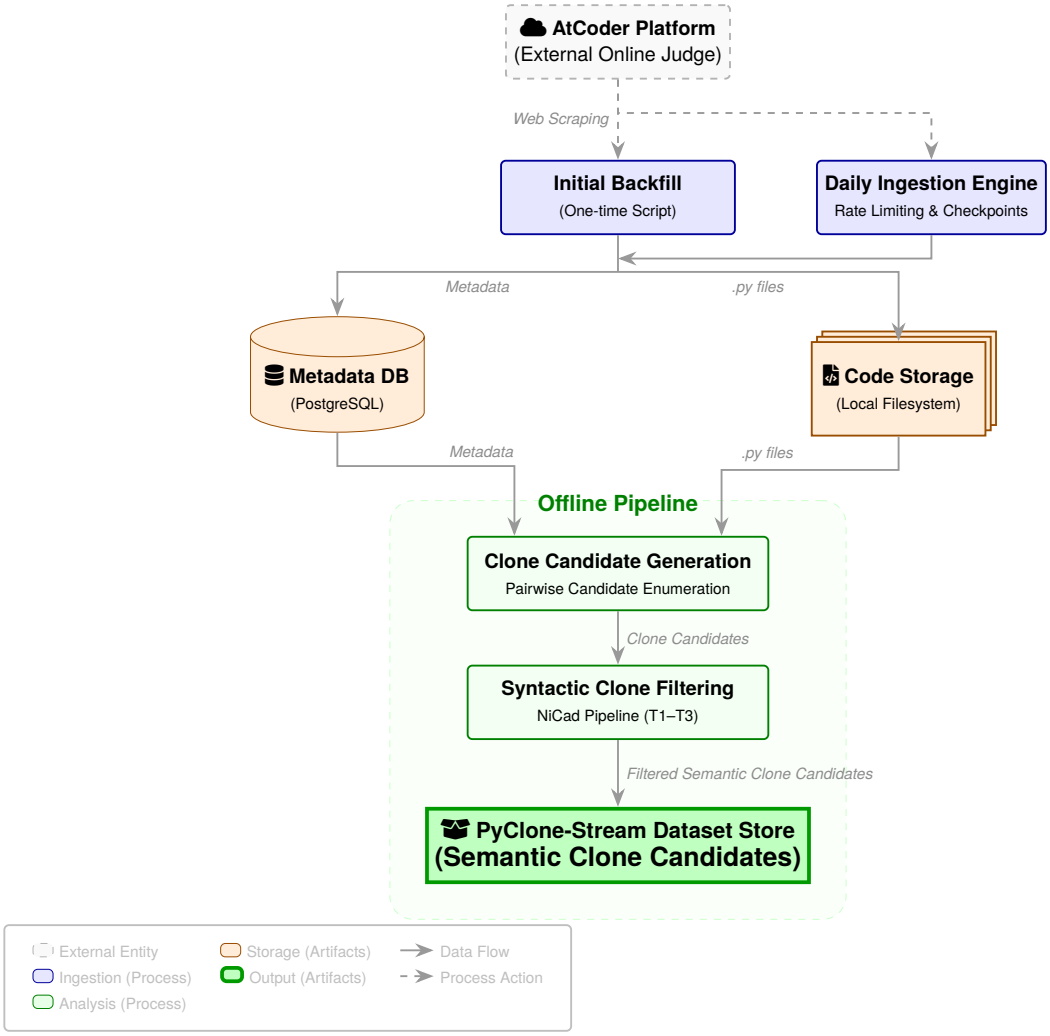

Fig. 1: PyClone-Stream pipeline architecture

design styles emerge over time. These trends can be easily observed on competitive programming and online coding platforms that serve as major sources of real-world code. As a result, static datasets risk becoming outdated, limiting their viability for testing modern learning-based models. Moreover, beyond issues of scale and staleness, recent analyses show that even well-established datasets can lead to misleading conclusions when their labeling assumptions are misapplied or insufficiently examined [6]. Taken together, these factors elevate dataset construction from a one-time preprocessing step to a central research challenge.

2.1 Problem Statement

The absence of a large-scale, reproducible dataset that captures real-world semantic diversity in Python code hinders the objective evaluation and improvement of Type-4 clone detection strategies. Furthermore, it complicates the comparison of learning-based approaches under realistic conditions and the analysis of how evolving code distributions impact model performance.

To address this gap, we formalize the problem studied in this work. Specifically, the objective is to design a framework for constructing an automated, continuously growing, and reproducible dataset for Python semantic code clone detection that:

- captures real-world semantic diversity beyond syntactic similarity,
- scales over time without relying on manual annotation, and
- makes its labeling assumptions explicit and verifiable for downstream analysis.

Rather than treating dataset construction as a one-time curation effort, this work formulates it as a streaming and incremental process in which new Python code submissions are continuously collected, filtered, and merged into a growing corpus of candidate semantic clone pairings.

3. Methodology

In this section, we describe the architecture behind the PyClone-Stream pipeline, which ranges from automated data collection by web scraping to semantic clone labeling and dataset maintenance. The pipeline is designed to be highly automated, reproducible, scalable, and resilient to interruptions, enabling continuous dataset growth with minimal manual intervention.

Figure 1 illustrates the complete PyClone-Stream pipeline. The system comprises three loosely coupled stages: (1) automated data ingestion, (2) clone filtering and labeling, and (3) dataset expansion. This paper focuses on all three stages, which together define the dataset construction process.

3.1 Platform Choice for Data Collection

Code clones are code fragments that share some degree of syntactic or semantic similarity. On competitive programming platforms, developers worldwide attempt to solve the same problem by using different algorithms, design choices, and approaches. This makes such platforms a natural testbed for collecting code clones, as developers essentially create code clones for the same problem. In this work our choice is AtCoder[(1)] competitive programming platform.

3.2 Automated Contest and Submission Collection

Contest Selection Strategy. PyClone-Stream primarily targets recent contests on the AtCoder platform to capture the latest changes and updates in software engineering, with older contests serving as a backup. Contests on AtCoder contain a wide range of difficulty levels of problems. Contests are divided into four types according to the difficulty level of the problems they contain. In order to make the dataset diverse in terms of problems, the following strategies are followed:

- **Collection window**: 20 contests collected over a fixed period from February 2025 to January 2026.
- **Booster sampling**: Older contests serve as a backup in case the problems of the recent contests do not contain enough submissions.

This approach allows for the tracking of recent trends while preserving temporal relevance.

Problem and Submission Filtering. There is no restriction on the number of the problems per contests. Our pipeline scrapes all the problems from the contest details page, as contests typically contain a limited number of problems. For submissions, we retrieve only Python submissions with an "Accepted" status to ensure that the dataset contains functionally correct code samples.

We also enforce a per-user-per-problem policy in our pipeline to reduce redundancy and author-specific bias. Specifically, at most k submissions per user are retained for each problem, where k is a configurable parameter (set to 10 in the current implementation).

Rate Limiting and Fault Tolerance. To ensure reliable and responsible data collection from the AtCoder platform, the scraping pipeline incorporates standard rate limiting and fault-tolerance mechanisms. Requests are throttled to comply with server usage constraints, and transient failures are handled using retry and backoff strategies. Additionally, lightweight checkpointing is employed to enable safe recovery from interruptions, allowing the pipeline to resume data collection without reprocessing previously handled data.

3.3 Data Collection and Data Storage

There are two operational modes in PyClone-Stream:

Initial Backfill. The initial Backfill mode is executed once when deploying the system. Its purpose is to crawl all eligible contests within the configured time frame.

Incremental Updates. After the initial backfill, a daily cron job retrieves newly available contests, problems, and submissions. Since all operations are idempotent and checkpointed, incremental updates can be performed safely without reprocessing previously collected data. Due to this design, the system consistently resumes from the last processed contest and submission page.

The collected data are stored in the following forms:

- **Metadata Storage:** All collected metadata such as details of contests, problems, users, language versions and submissions are stored in a normalized PostgreSQL database. Each table includes timestamp fields (`created_at`, `updated_at`) to support temporal analysis and dataset auditing.
- **Code Storage:** To avoid database bloat, source code is stored separately from metadata details in a filesystem-based repository. Each submission is written to a deterministic path based on platform, contest, problem, and submission identifier. These paths are stored in the database in the corresponding table and column. Alongside these paths cryptographic hash (SHA-256) per file is generated to detect duplication and ensure integrity. Other reasons for designing the architecture this way are to simplify backups, and enable efficient access by external analysis tools such as *NiCad* [9].

(1): `https://atcoder.jp`

Algorithm 1 Syntactic Clone Candidate Extraction

```
for each problem P do
    collect submissions S_P
    generate pairs C_P = {(s_i, s_j)}
    run NiCad on C_P
    remove detected Type-1–3 clone pairs
    label remaining pairs as Type-4 candidates
end for
```

3.4 Clone Candidate Generation

For each problem P, let $\mathcal{S}_P$ denote the set of accepted Python submissions collected. PyClone-Stream generates all unordered pairs (s_i, s_j) where $s_i, s_j \in \mathcal{S}_P$ and $i < j$. Each pair represents a potential clone candidate.

On the AtCoder platform, problems are categorized by difficulty level. The higher the alphabetic label, the more difficult the question is. In our pipeline, we manage the risk of combinatorial explosion by assigning caps to the number of submissions processed per problem. Since our purpose is to create a code clone dataset at the function granularity, simpler problems (e.g., ABC A/B) are assigned higher caps, as they are more likely to yield single-function solutions. In contrast, harder problems (e.g., D/E/F) are assigned lower caps.

3.5 Syntactic Clone Filtering and Labeling

Rationale. Since many generated clone pairs from collected submissions have the potential to be Type-1, Type-2, and most Type-3 (i.e., syntactic) clones, we cannot treat all submissions as semantic clones as this would introduce substantial noise. Instead, PyClone-Stream adopts a conservative filtering strategy in which a traditional code clone detector is used to isolate semantic clones from syntactic clones. In our use case, NiCad is chosen for this purpose since it is one of the most reliable and widely used code clone detection tools for syntactic clones. After this filter, we treat the remaining pairs as Type-4 clone candidates, under the assumption that traditional clone detection tools achieve sufficiently high recall for Type-1 to Type-3 clones within the considered code corpus. Consequently, pairs surviving this filtering step are unlikely to exhibit purely syntactic or near-miss structural similarity.

Filtering Algorithm. Algorithm 1 presents the filtering process. The objective of the filtering step is to eliminate syntactically similar clone pairs (Type-1-3) from the generated candidate set and retain only Type-4 clone candidates. For each problem P, the algorithm takes as input the set of accepted submissions $\mathcal{S}_P$ and the corresponding submission pairs generated from it. NiCad is applied to these pairs to detect Type-1-3 clones under a fixed configuration. All detected syntactic clones are removed, and the remaining pairs are labeled as Type-4 clone candidates for downstream analysis.

NiCad Configuration. NiCad is configured to detect clones under multiple normalization settings, including identifier renaming and literal abstraction. Thresholds are chosen conservatively to favor

precision over recall. In our use case, the dissimilarity threshold is set to 0.7, ensuring that syntactically similar code fragments are reliably filtered out.

Labeling Assumptions and Noise. After syntactic clone filtering, PyClone-Stream labels the remaining clone candidates as Type-4 code clone candidates under the assumption that accepted submissions to the same problem are functionally equivalent. While this assumption introduces some label noise, it allows scalable dataset construction without human-in-the-loop annotation.

Dataset Evolution and Reproducibility. All scripts, configurations, and dataset schemas are designed to support reproducibility and extensibility, and continuous dataset growth.

3.6 Automation and Operational Considerations

Once the project is deployed, it performs data scraping and clone curation autonomously. While human intervention is not required during routine operation, rare maintenance actions such as manually renewing expired authentication credentials (e.g., `revel_session`) are required and this is dictated by platform access policies rather than pipeline design.

4. Dataset Construction and Statistics

In this section, we describe the composition and statistical characteristics of the PyClone-Stream dataset in the current snapshot.

PyClone-Stream is a continuously growing dataset of semantic code clone candidates derived from accepted competitive programming submissions that are written in Python. The results reported in this paper correspond to a snapshot collected after the initial backfill.

4.1 Raw Submissions

The current snapshot contains accepted Python submissions collected from AtCoder contests. After filtering for language, acceptance status, and per-user submission limits, the dataset includes:

- 20 contests
- 120+ problems
- 22,000+ Python submissions
- 0.1M+ Type-4 candidate pairs

The source code of each submission is stored as a file under a specific directory and linked to metadata including platform ID, contest ID, problem ID, submission ID, language ID, status, code path, SHA-256 cryptographic hash of the code, and code size in bytes.

4.2 Problem-Level Distribution

All submissions are grouped by problem, where each group consists of multiple independent implementations of the same programming task. Problems with fewer than two submissions are excluded. Across retained problems:

- The average number of submissions per problem is 190
- The largest group contains 607 submissions
- The smallest group contains 2 submissions

This grouping forms the basis for clone pair generation.

4.3 Clone Candidate Pairs

For each problem with n submissions, all unordered submission pairs are generated, yielding $n(n-1)/2$ candidate pairs. The current snapshot contains more than 0.1M Type-4 clone candidates. All pairs originate from submissions that solve the same problem and therefore share functional intent.

4.4 Syntactic Clone Removal

Syntactic clone filtering substantially reduces the number of candidate pairs and reshapes the dataset composition. After applying NiCad-based filtering, all detected syntactic clones are removed. Approximately 2.4 million syntactic clone pairs are filtered out. The remaining pairs demonstrate significantly greater syntactic and structural diversity and constitute the pool of semantic (Type-4) clone candidates retained in PyClone-Stream.

4.5 Dataset Contents

Each clone pair in PyClone-Stream includes:

- Identifiers of both submissions
- Associated problem and contest identifiers
- Cloning Metrics

This dataset does not include manual annotations; instead, labels are derived automatically based on problem equivalence and clone filtering.

4.6 Dataset Comparison

Table 1 highlights key differences between PyClone-Stream, SemanticCloneBench [10], and GPTCloneBench [11]. In contrast to SemanticCloneBench, which is a static benchmark constructed once, PyClone-Stream is designed as a continuously growing dataset with incremental daily ingestion. While SemanticCloneBench and GPTCloneBench provide curated semantic clone pairs, their scale is limited or dependent on model-driven generation, respectively. PyClone-Stream uniquely derives clone candidates from organically written, human-authored competitive programming submissions, enabling substantially larger volumes of Type-4 clone candidates to be collected on a recurring basis. Moreover, unlike GPTCloneBench, which relies on large language models for clone generation or labeling, PyClone-Stream employs a conservative, tool-based filtering strategy, prioritizing reproducibility and minimizing model-induced bias. These characteristics position PyClone-Stream as a scalable and extensible dataset infrastructure for long-term analysis and learning-based semantic clone detection.

5. Illustrative Examples

To clarify the distinction between syntactic and semantic code clones in the context of PyClone-Stream, we present representative examples drawn from real accepted submissions on the AtCoder competitive programming platform. All examples correspond to correct Python solutions for the same problem and are included for illustrative purposes only.

5.1 Problem Definition

The task considered in the following examples is defined as follows:

> You are given a positive integer X. Find the minimum value among all positive integers that can be obtained by rearranging the digits of X, subject to the constraint that the resulting number must not contain leading zeros.

This problem admits multiple correct solutions, allowing developers to employ different algorithmic strategies while preserving functional correctness.

5.2 Semantic Clone Example (Type-4)

Figure 2 presents two accepted solutions, denoted as (a) and (b), that solve the same problem using substantially different implementation strategies. Solution (a) explicitly separates zero and non-zero digits and constructs the result through sorting and list manipulation, whereas solution (b) reconstructs the output using digit frequency counts.

```
X = input()
c = X.count('0')
S = []
for i in X:
    if i != '0':
        S.append(i)
S.sort()
print(*([S[0]] + ['0'] * c + S[1:]), sep="")
```

(a) Solution (a)

```
X = input()
count = [0 for _ in range(10)]
n_min = 9
for i in X:
    count[int(i)] += 1
    if int(i) > 0:
        n_min = min(n_min, int(i))
ans = str(n_min)
count[n_min] -= 1
for i in range(10):
    ans += str(i) * count[i]
print(ans)
```

(b) Solution (b)

Fig. 2: Semantic clone pair candidate: structurally different implementations exhibiting identical behavior.

Despite these structural and algorithmic differences, both implementations produce identical outputs for all valid inputs and satisfy the problem constraints. Due to their semantic equivalence combined with substantial syntactic and structural divergence, this pair constitutes a *semantic (Type-4) clone pair*.

5.3 Filtered Syntactic Clone Example (Type-2)

Figure 3 presents a contrasting example consisting of solutions (a) and (c). Both implementations follow an almost identical syntactic structure: they count the number of zero digits, collect all non-zero digits into a list, sort the list, and construct the final output by placing the smallest non-zero digit first, followed by all zeros and the remaining digits. The differences between the two solutions are limited to superficial variations such as variable naming, loop syntax, and output formatting. Since the core control flow and data manipulation logic are preserved with only minor textual modifications, this pair is classified as a renamed syntactic clone (Type-2) and is correctly filtered out during preprocessing by NiCad.

6. Related Work

Progress in semantic (Type-4) code clone detection via learning-based approaches strongly depends on the availability of benchmark datasets. However, existing datasets continue to constrain their effectiveness and generalizability. In this section, we review prior work with a focus on semantic clone datasets and position PyClone-Stream within this landscape. Unlike Section 5, which presents qualitative examples between PyClone-Stream and existing datasets, this section situates PyClone-Stream within the broader research context and highlights methodological differences from prior work.

```
X = input()
c = X.count('0')
S = []
for i in X:
    if i != '0':
        S.append(i)
S.sort()
print(*([S[0]] + ['0'] * c + S[1:]), sep="")
```

(a) Solution (a)

```
s = input()
zero = s.count("0")
ans = []
for x in s:
    if x != "0":
        ans.append(x)
ans.sort()
print(ans[0]+"0"*zero+"".join(ans[1:]))
```

(b) Solution (c)

Fig. 3: Syntactic clone pair filtered out during preprocessing due to high structural similarity.

6.1 Code Clone Datasets

There are several datasets that are widely used in code clone research:

- *BigCloneBench* is one of the most influential datasets, containing a large collection of Java clone pairs annotated according to functional similarity [7]. It has played a foundational role in the development and evaluation of code clone detection techniques and is widely used as a benchmark dataset. However, BigCloneBench exhibits several important limitations. It is static and Java-centric, and a substantial portion of its clone pairs are syntactically similar and derived from a fixed snapshot of Java projects. As a result, the dataset provides limited support for analyzing the diversity and temporal evolution of real-world code, which may affect the generalizability of semantic code clone detection studies in more diverse or dynamic programming contexts.
- The *POJ-104* dataset focuses on algorithmic problems and provides labeled pairs of semantically equivalent programs [8]. While POJ-104 captures functional equivalence more directly, it is limited in scale, programming language diversity, and structural variability. Moreover, it is typically treated as a closed benchmark, offering no support for incremental growth or longitudinal analysis.
- *SemanticCloneBench* provides curated benchmarks specifically designed for evaluating semantic code clone detection [10]. However, the dataset is relatively limited in scale, containing 1,000 Python semantic clone pairs out of 4,000 pairs across four programming languages, which restricts its coverage and applicability.
- *GPTCloneBench* explores the use of large language models to

Table 1: Comparison between PyClone-Stream and existing semantic clone benchmarks.

Aspect	PyClone-Stream	SemanticCloneBench	GPTCloneBench
Dataset Nature	Continuously growing (Streaming)	Static benchmark	LLM-Generated/Labeled
Primary Lang.	**Python**	C, C#, Java, Python	Multiple (Model-dependent)
Data Source	Competitive programming (AtCoder)	Stack Overflow snippets	Synthetic / LLM-labeled
Scale	$>$ 20**K** submissions; $>$ 0.1**M** Type-4 clone candidates*	4,000 pairs (1,000 are Python)	Variable; depends on LLM sampling
Update Model	Daily incremental updates	One-time construction	On-demand generation

*Scale based on initial backfill; capable of autonomous daily growth.

generate or label semantic clone pairs [11]. While this approach enables rapid dataset construction, it introduces potential biases associated with model-generated code, prompt sensitivity, and reliance on model behavior rather than organically written human code.

These limitations motivate the design of PyClone-Stream, which emphasizes continuous growth, scalability, and the use of real-world, human-authored code to better capture the diversity and evolution of semantic clones.

6.2 Filtering-Based Dataset Construction

One common use of traditional clone detection tools is as filtering mechanisms in dataset construction to isolate semantically equivalent clone pairs. In particular, tools such as NiCad can detect Type-1, Type-2, and many Type-3 clones with high precision through configurable normalization strategies and similarity thresholds [9]. Because semantic equivalence is undecidable in the general case, several studies adopt a residual-filtering strategy in which syntactic clones are removed before treating the remaining pairs as semantic clone candidates. SemanticCloneBench follows this approach by using NiCad to filter out syntactic clone pairs, while GPTCloneBench combines traditional filtering with large language models to generate or label semantic clone pairs. Following this line of work, PyClone-Stream adopts NiCad as a conservative syntactic filter to remove Type-1 to Type-3 clones, thereby enabling scalable construction of semantic clone candidates while making the underlying labeling assumptions explicit.

6.3 Positioning of PyClone-Stream

PyClone-Stream differs from existing code clone datasets in several key aspects.

First, it is designed as a continuously growing dataset, supporting automated and incremental data collection over time. To the best of our knowledge, PyClone-Stream is one of the first code clone datasets explicitly designed to be self-growing.

Second, PyClone-Stream focuses on Python, a dynamically typed language that, despite its widespread use, remains underrepresented in existing code clone datasets.

Third, the dataset is constructed from organically written, human-authored code collected from competitive programming platforms, providing diverse real-world implementations rather than model-generated or manually curated code fragments.

Fourth, the pipeline is designed to be scalable, reproducible, and easily extensible to additional programming languages and platforms with minimal engineering effort.

Finally, code clone curation and data ingestion are largely automated. The pipeline operates without human intervention during normal execution, except for occasional maintenance tasks such as refreshing expired authentication sessions (e.g., `revel_session`). Taken together, these characteristics position PyClone-Stream not merely as another clone dataset, but as a continuously growing and automated dataset construction framework designed to support long-term research on semantic code clone detection.

7. Conclusion and Future Work

PyClone-Stream introduces a scalable framework for constructing semantic code clone datasets in Python. Future work will extend the pipeline to additional platforms and programming languages and leverage the continuously growing dataset to train, fine-tune, and evaluate learning-based models for semantic clone detection.

Acknowledgements

This research was supported by JSPS KAKENHI Japan (JP25K03102, JP24H00692, JP23K24823). The authors thank AtCoder Inc. for permitting data collection from their platform and for providing publicly accessible problem sets. All data were collected from the AtCoder platform using rate-limited, non-disruptive scraping.

References

[1] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, Stefan Wagner, “Do Code Clones Matter?,” in Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, pp. 494-495, 2009

[2] Chanchal K. Roy and James R. Cordy. “A Survey on Software Clone Detection Research,” Technical Report No. 2007-541, School of Computing, Queen’s University at Kingston, pp. 14, 2007.

[3] Zhangyin Feng et al., “CodeBERT: A Pre-Trained Model for Programming and Natural Languages,” in Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp.1536-1547, 2020.

[4] Daya Guo et al., “GraphCodeBERT: Pre-training Code Representations with Data Flow,” in Proceedings of the International Conference on Learning Representations (ICLR), 2021.

[5] Yue Wang et al., “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 (EMNLP), pp. 8696-8708, 2021.

[6] Jens Krinke, Chaiyong Ragkhitwetsagul, “How the Misuse of a Dataset Harmed Semantic Clone Detection,” arXiv preprint arXiv:2505.04311, 2025

[7] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, Mohammad M. Mia, “Towards a Big Data Curated Benchmark of Inter-Project Code Clones,” in Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2014.

[8] Lili Mou, Ge Li, Lu Zhang, Tao Wang, Zhi Jin, “Convolutional Neural Networks over Tree Structures for Programming Language Processing,” in Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI), pp. 1287-1293, 2016.

[9] James R. Cordy, Chanchal K. Roy, “The NiCad Clone Detector,” in Proceedings of the 19th IEEE International Conference on Program Comprehension (ICPC), 2011.

[10] Farouq Al-Omari, Chanchal K. Roy, Tonghao Chen, “SemanticCloneBench: A Semantic Code Clone Benchmark using Crowd-Source Knowledge,” in Proceedings of the International Workshop on Software Clones (IWSC), 2020

[11] Ajmain I. Alam et al., “GPTCloneBench: A comprehensive benchmark of semantic clones and cross-language clones using GPT-3 model and SemanticCloneBench,” In Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2023.