Reimplementing and Optimising Pyttern in Java for Faster Pattern Matching

Julien Liénard[1,*], Quentin Lépine[2] and Kim Mens[1]

[1]UCLouvain, ICTEAM, Louvain-la-Neuve, Belgium
[2]UCLouvain, EPL, Louvain-la-Neuve, Belgium

Abstract

Pyttern is a pattern matching language for Python, to describe and match code patterns in a syntax close to Python, extended with wildcards. One of its use cases is to analyse simple student programs, to provide targeted feedback on common coding idioms and flaws. Performance limitations of Pyttern's current implementation hinder its use at scale, especially for more complex patterns over larger datasets of student programs. This paper tackles two key performance bottlenecks: parsing speed and combinatorial pattern matching. By reimplementing Pyttern in Java, parsing time is reduced by a factor 20 on large datasets, thanks to Java's just-in-time compilation. In addition, we optimize the matching algorithm by removing duplicate-producing transitions, pruning redundant wildcards, and introducing path sharing to avoid repeated computations. These algorithmic improvements further reduce the cost of combinatorial patterns, significantly enhancing Pyttern's scalability and practical usability.

Keywords

Pyttern, Pattern Matching, Python, Programming, Java, Benchmarking, Reimplementation, Optimisation

1. Introduction

Pyttern [1] is a pattern matching language for Python, with a syntax close to Python, extended with expressive wildcards to leave some parts of the program unspecified. As such, Pyttern patterns become structural templates of the set of programs they are intended to match, i.e. each program that contains the structure specified by the Pyttern pattern, but that can vary in the unspecified parts.

In software evolution and maintenance, program query languages (PQLs) serve as a foundational mechanism for tasks such as automated code linting, bug and smell detection, and enforcing architectural or stylistic rules. Under the hood of developer tools, such pattern matchers must operate with low latency—either interactively within developer IDEs or across entire code repositories in continuous integration (CI) pipelines. A particularly demanding instance of this workload arises in educational environments, where linters and query engines analyze submissions from large cohorts in introductory programming courses to provide automated feedback on recurring idioms and misconceptions.

In such high-throughput scenarios, Pyttern's current implementation in Python suffers from speed bottlenecks that disrupt the user's workflow. An initial profiling study, based on six representative patterns, showed that the mere parsing of input programs dominated the runtime cost in five cases, accounting for over 97% of the total runtime cost. For the sixth case the matching itself was the bottleneck, because the wildcards used in that pattern forced the matcher to explore a combinatorially growing number of cases. This paper aims to overcome both bottlenecks, by reimplementing Pyttern in Java and further optimizing it, without altering its syntax or matching semantics. As a tool and experience paper, this work contributes both practical engineering insights into migrating AST query runtimes and algorithmic solutions to eliminate combinatorial non-determinism in tree-automata matching. Specifically, our contributions are:

BENEVOL2026, November 26-27, Madrid, Spain
*Corresponding author.
julien.lienard@uclouvain.be (J. Liénard); quentin.lepine@student.uclouvain.be (Q. Lépine); kim.mens@uclouvain.be (K. Mens)
0009-0009-2966-7179 (J. Liénard); 0000-0003-0303-1630 (K. Mens)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

- **Algorithmic reductions of tree-automaton non-determinism:** We introduce three complementary techniques that eliminate combinatorial state-space explosion: (i) correcting the pushdown automaton (PDA) encoding of the `?:*` wildcard using stack-top guards to eliminate an exponential duplication of exploration branches; (ii) defining parse-tree rewriting rules that collapse adjacent wildcards before automaton construction; and (iii) designing a path-sharing matcher that merges convergent search configurations via path counting, avoiding duplicate sub-tree traversals while preserving exact match counts and variable bindings.
- **Empirical characterisation and scalability analysis:** We conduct an in-depth empirical study profiling realistic query workloads, isolating parsing as the 97%–99% common-case bottleneck. Furthermore, through differential benchmarking across file-size and batch-size workloads—including a JIT-disabled (`-Xint`) configuration—we demonstrate that HotSpot's adaptive compilation, rather than ANTLR library maturity, is the primary driver of the up to 20× speedup. Finally, we evaluate the interaction and interference among the three matching optimisations.
- **A production-ready tool reimplementation:** We provide a complete, open-source Java reimplementation of Pyttern preserving the original ANTLR grammars and matching semantics, together with a replication package for the empirical benchmarks.

The paper is organised as follows. Section 2 introduces the Pyttern language and its matching pipeline based on a pushdown automaton. Section 3 positions Pyttern with respect to alternative pattern-matching tools. Section 4 reports the profiling experiment that identifies parsing as the dominant cost, motivating the choice of a full Java port over a hybrid solution. Section 5 validates the first stage of the port by comparing the Java and Python parsers across file sizes and batch sizes. Section 6 validates the second stage by evaluating the full Java pipeline against the original Python implementation. Section 7 develops and evaluates the three matching optimisations and analyses their interference. Section 8 collects practical lessons learned during the port. Section 9 outlines two directions for further improvement: reducing the JVM startup overhead and replacing the parser by a faster alternative. Section 10 concludes the paper by synthesising the contributions and revisiting its objectives.

2. Pyttern

2.1. Syntax and Semantics

Pyttern is a dedicated pattern language for structural matching of Python source code [1]. Whereas existing program querying frameworks often require users to master complex query languages or to reason directly over syntax trees, Pyttern was designed to make structural source-code patterns easier to express. To achieve this, it keeps a syntax close to regular Python and merely adds a small set of regular expression-inspired wildcards. Pyttern's original Python implementation[1] and the Java reimplementation with the benchmarking replication suite[2] are both publicly available on GitHub.

The Pyttern pattern language can be viewed as an extension of the target programming language Python enriched with various wildcards. Pyttern provides seven kinds of wildcards, each corresponding to a distinct matching behaviour, such as matching a single syntactic element, a sequence of elements, or an indented body. For instance, the wildcard `?name` matches any element on its first occurrence. Subsequent occurrences of the same named wildcard must match that same element, as illustrated in Table 1 where the pattern `?x` will be matched twice against the *same* literal a.

Table 2 presents a more advanced example. The pattern matches code containing a hardcoded list which can be used for example to detect students who store prime numbers in a list instead of computing them dynamically. The function header on line 1 uses the wildcard `?` to match any function name and `?*` to match any parameter list, while the `?:*` on line 2 allows the assignment below to appear at any nesting level within the function body. The assignment itself, on line 3, matches some variable

[1] https://github.com/JulienLie/Pyttern
[2] https://github.com/Qlepine/Pyttern

Code	Pattern

`def foo():`
`  a = 0`
`  return a`
	`def ?():`
	`  ?x = 0`
	`  return ?x`

Table 1
Wildcard matching example.

(wildcard `?`) assigned to a list literal whose length is constrained to at least three elements by `?{3,}`. The pattern therefore captures any function that, somewhere in its body, assigns a list of three or more elements to a variable, regardless of the function signature.

Code	Pattern

`def factors(n):`
`  l = [2,3,5,7,11,13]`
`  t = []`
`  for i in range(len(l)):`
`  if n%l[i] == 0:`
`  t.append(l[i])`
`  i -= 1`
`  nbr = len(t)`
`  return (nbr)`
	`def ?(?*):`
	`  ?:*`
	`  ? = [?{3,}]`

Table 2
Hardcoded list matching example.

A brief description of all these wildcards is provided in Table 3.

Wildcard Type	Pyttern	Semantics
Simple	`?`	Match any single syntax tree element
Named	`?var`	Match and bind an element by name
List	`?*`	Match 0 or more elements in a list
Number	`?{n,m}`	Match between n and m elements
Simple Compound	`?:`	Match one level of indentation/block nesting
Multiple Compound	`?*:`	Match arbitrary indentation levels
Contains	`?< >`	Match subtree containment in nesting expressions

Table 3
Syntax and Semantics of Pyttern Wildcards

2.2. Pattern Matching

As depicted in Figure 1, a Pyttern expression is matched against Python code in three main steps [1].

1. Parsing First, both the Python code and the pattern are parsed to obtain a tree representation. Pyttern relies on ANTLR, a parser generator, to build a parse tree from a given grammar. The grammar depends on the targeted language: in practice, one grammar is used for the source language (e.g., Python) and one extended grammar is used for the pattern language (i.e., the same language extended with wildcard constructs). The raw parse trees obtained are then refined by a dedicated visitor that discards non-semantic tokens and flattens grammatical structures that are more detailed than the matching process requires. The resulting tree is composed of typed nodes defined by the grammar (Block node

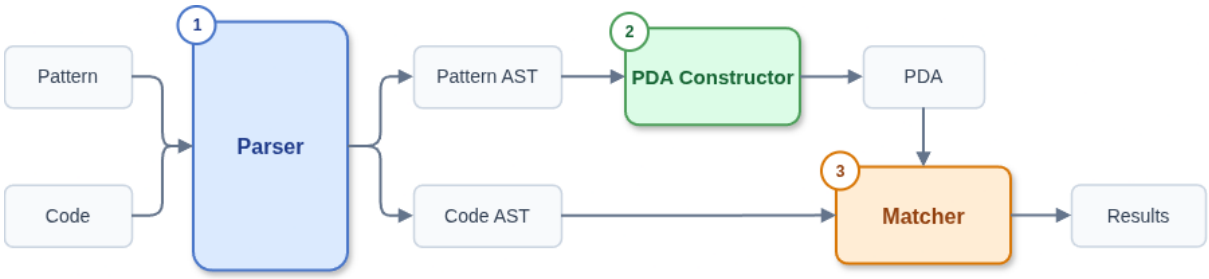


Figure 1: High-level workflow of Pyttern.

for indented body, Stmt node for statement in Python, etc.); these types are used later to condition the transitions of the pushdown automaton. Although the current implementation of Pyttern only supports Python, in principle, any language for which a grammar is available could be supported, provided that the grammar and the resulting tree representation are adapted to integrate wildcards.

2. Pushdown automaton construction Second, based on the pattern's tree, Pyttern builds a pushdown automaton (PDA), which serves as an executable representation of the pattern used by the matcher to navigate and match the Python code tree. The PDA is a finite-state machine augmented with a stack, which allows it to handle the nested, hierarchical structure of a parse tree. The PDA builder translates each node of the pattern tree into PDA transitions, with each type of wildcard producing its own specific transition pattern; a list wildcard, for example, generates a self-loop to consume arbitrarily many siblings. In Pyttern, the stack uses two symbols: `I` is pushed each time the matcher descends into a child node and popped when it returns to the parent, while `B` is pushed on entry to a wildcard body (`?:*` or `?<...>`) and popped on exit, marking the boundary of the region matched by the wildcard. The stack therefore reflects both the chain of descents between the root of the source tree and the current position and the boundaries of regions matched by wildcard bodies, allowing the automaton to take decisions that depend on the surrounding context. By visiting the full pattern tree, the PDA provides a structured way to match the pattern against the code tree.

Each transition from a state q_i to a state q_j in the PDA carries five pieces of information, summarised in Figure 2: a condition on the top of the stack (a symbol to read, for example a `B` symbol at the top of the stack), a condition on the current node of the source code (its type, or its text in the case of a terminal node; in the figure, the node must be of type `If_stmt`), an instruction to navigate in the source tree (such as moving to the left child, the right sibling, or the parent; `LEFT_CHILD` in the figure), a destination state (q_j in the figure), and new symbols to push onto the stack (`B` and `I` in the figure). A transition can only be taken when both the stack and the current node match its conditions. This combination of stack and tree conditions is what makes the PDA capable of expressing the structural constraints encoded in a pattern.

3. Matching During the matching process, Pyttern traverses the Python code tree guided by the transitions of the pushdown automaton. Because wildcards allow several transitions to apply from the same point, the exploration is not limited to a single path, and the matcher therefore maintains a list of configurations, each representing a snapshot of one ongoing exploration of the pattern against the program code: a state, a position in the source tree, a stack, and a set of variable bindings. At each step, the matcher takes one configuration from the list, computes every transition that is applicable from it, and adds the resulting new configurations back to the list; this process continues until no configuration remains. Several transitions can be applicable from the same configuration, in which case the exploration splits into as many branches, all carried out in parallel. Whenever a configuration reaches the acceptance state, it is recorded as a match.

Depending on the use case, Pyttern supports both *first-match* exploration (halting as soon as an

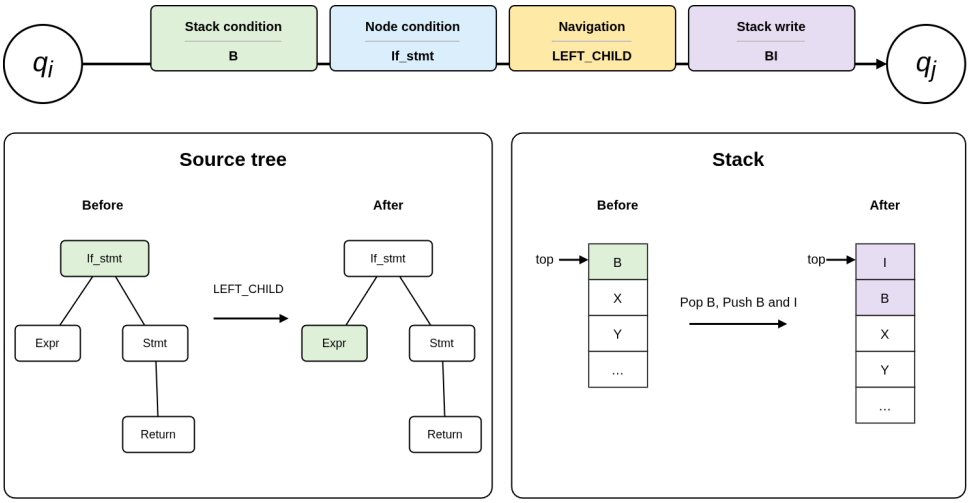


Figure 2: Anatomy of a PDA transition in Pyttern.

acceptance state is reached, which is ideal for fast existential or boolean queries) and *all-matches* exploration. Unlike term rewriting systems where matching is typically restricted to eager first-match reduction, Pyttern's primary use case is program querying and static code linting. In these scenarios, the goal is not merely to check whether an idiom or defect exists, but to report *all* occurrences, AST locations, and distinct variable bindings across the analyzed code (e.g., detecting every hardcoded list or every flawed recursive call in a student submission). Halting at the first match would yield incomplete diagnostics. Consequently, when performing exhaustive matching, the resulting combinatorial growth of configurations cannot be bypassed by early termination. This growth is the primary bottleneck tackled by the algorithmic optimisations presented in Section 7.

3. Related Work

Program querying frameworks and pattern-matching tools for source code generally fall into two broad categories: general-purpose program query formalisms operating directly over intermediate representations, and high-level pattern languages that mimic concrete target language syntax.

Program Query Formalisms and AST Querying Declarative meta-programming approaches such as SOUL [2] reify source code into logic clauses to explicitly codify and search for programming patterns via logic queries. Formalisms such as PQL [3], codeQuest [4], and Rascal [5] provide powerful abstractions for program analysis, program transformation, and bug detection. PQL uses context-free matchers and relational queries, while codeQuest maps source code into relational databases queried with Datalog [6]. Rascal combines meta-programming, pattern matching, and rewriting rules in a specialized domain-specific language. While highly expressive, these tools require users to master complex query dialects and reason directly over relational abstractions or AST data structures, creating a steep learning curve for non-experts.

Concrete Syntax and Unparsed Pattern Matching To reduce the entry barrier, several tools allow pattern specification close to source code syntax. Volanschi et al. [7] introduced unparsed patterns to enable user-extensible code transformations without full syntactic specification. Modern lightweight tools such as Semgrep leverage concrete code syntax enriched with wildcards to make pattern creation accessible to software engineers.

Pattern Matching in Educational Tooling In introductory programming education (CS1), pattern-matching tools have been developed to detect common student misconceptions and provide automated

feedback. Harden et al. [8] developed a dedicated specification language to match mistake patterns and map them to targeted pedagogical feedback. Gusukuma et al. [9] introduced Pedal, a modular feedback infrastructure integrating structural program analysis. Recent empirical evaluations by Colla et al. [10] compare the trade-offs of different program query languages for detecting Python student misconceptions, underscoring the critical need for scalable execution on large student cohorts.

Positioning of Pyttern Pyttern [1, 11] bridges the gap between concrete syntax accessibility and structural expressiveness. By extending standard Python syntax with intuitive wildcards backed by a pushdown automaton (PDA) over parse trees, Pyttern enables instructors to write structural queries without learning specialized formalisms. Reimplementing and optimizing Pyttern in Java preserves this accessibility while resolving the performance bottlenecks that previously limited its deployment over large student program datasets.

4. Profiling

To identify the primary performance bottlenecks in the original Python implementation of Pyttern, we conducted a profiling benchmark under realistic usage conditions.

4.1. Benchmark Setup

We evaluated the execution of correct student submissions against representative Pyttern patterns, recording per-stage and end-to-end execution times. The dataset consisted of 50 student solutions to a real-course programming exercise and 6 patterns from the Pyttern test suite, resulting in 300 (submission, pattern) pairs.

Rather than relying on synthetic or trivial patterns—which finish quasi-instantaneously and mask runtime bottlenecks—these 6 patterns were chosen according to three deliberate criteria:

1. **Wildcard diversity:** They collectively exercise the full range of Pyttern's wildcard constructs, including single-node wildcards (`?`), sequence wildcards (`?*`), repetition bounds (`?{n,m}`), block nesting (`?:`), and arbitrary indentation depth (`?:*`).
2. **Real-world pedagogical idioms:** They represent genuine structural patterns identified in student codebases for automated feedback: detecting hardcoded lookup tables instead of dynamic computation (`hardcodedlist`), direct recursion (`recursion`), deep nesting anti-patterns (`indentation`), accumulator loops (`piPattern`), conditional search loops (`pattern13`), and listener registration (`observer`).
3. **Structural edge cases:** They vary significantly in AST depth and branching potential, contrasting predictable, linear AST navigations with patterns that trigger combinatorial exploration.

A brief summary of each pattern and its primary wildcards is provided in Table 4.

Each pair was benchmarked using `pyperf` [12] (3 warm-up iterations, 5 timed iterations per worker process). Internal caches were cleared before each run to measure the raw cost of ANTLR lexing and parsing.

All benchmarking experiments across this paper were conducted on a dedicated virtual machine running Ubuntu 24.04.4 LTS (Linux kernel 6.8.0-88-generic, `x86_64`), configured with 4 vCPUs and 4 GB of RAM. The Python benchmarks were executed using Python 3.12.3, and the Java benchmarks using OpenJDK 25.0.2.

4.2. Results and Bottleneck Analysis

The profiling results in Figure 3 demonstrate that for 5 out of the 6 evaluated patterns, parsing is the overwhelming bottleneck, accounting for between **97.81%** (`indentation`) and **99.95%** (`observer`) of total execution time. In these cases, matching accounts for only 0.046%–2.19% of the runtime,

Pattern	Targeted Programming Idiom / Defect	Key Wildcards
`observer`	Observer class with listener registration and notifications	`?`, `?*`, `?name`
`recursion`	Recursive function call binding function name	`?name`, `?*`
`indentation`	Function body containing three nested indented blocks	`?:`, `?:*`
`hardcodedlist`	Variable assigned to a hardcoded list of ≥ 3 elements	`?`, `?*`, `?{3,}`, `?:*`
`pattern13`	Loop over a range with a conditional zero-test (`if ... == 0`)	`?`, `?*`
`piPattern`	Accumulator loop computing a series approximation	`?`, `?*`

Table 4
Patterns used in the profiling benchmark.

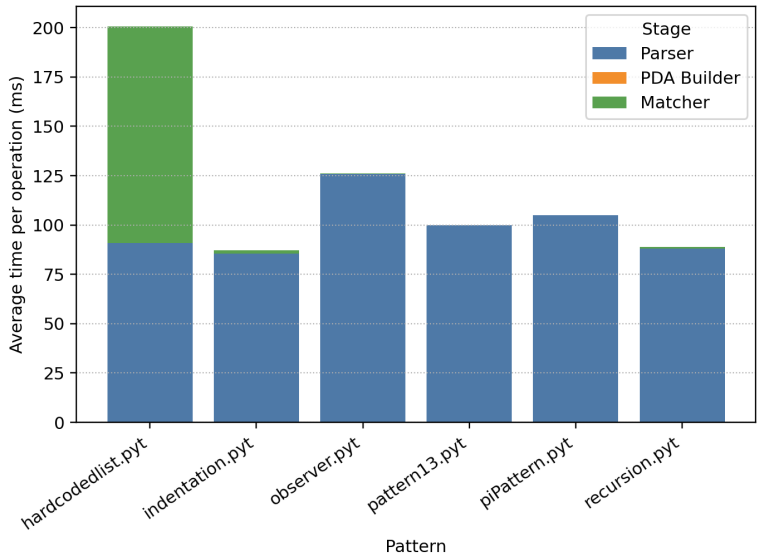


Figure 3: Average execution time per stage for each pattern (mean across 50 submissions of the per-pair `pyperf` mean).

while PDA construction cost remains negligible ($< 10^{-4}\%$). End-to-end execution times ranged from approximately 88 ms to 203 ms per submission.

The single exception was `hardcodedlist`, where parsing represented **44.07%** of runtime and matching accounted for **55.93%**. This shift occurred because wildcards accepting zero or more elements (such as `?*`) forced the pushdown automaton to explore multiple syntactic decompositions, causing search tree branching during matching.

4.3. Motivation for a Java Reimplementation

Since parsing dominates execution time in most cases, optimizing parsing yields the highest global performance impact. We opted for a full Java port over a hybrid architecture to avoid cross-language invocation and serialization overheads. We hypothesized that Java would provide substantial speedups due to the maturity of its ANTLR runtime and HotSpot's adaptive Just-In-Time (JIT) compilation. The port was structured into two stages: Stage 1 reimplements the parsing stage in Java, and Stage 2 reimplements the complete pipeline (PDA construction and matching engine).

5. Parsing Validation

This section validates Stage 1 of the port by evaluating the performance of the Java parser against the Python parser across file-size and batch-size workloads.

5.1. Methodology

To compare both implementations under identical conditions, we measure end-to-end execution time externally using `hyperfine` [13]. Each run invokes a fresh process, representing a cold-start scenario where JVM initialisation overhead is fully included.

To isolate the source of performance gains, we also evaluate a complementary configuration: running the Java parser with Just-In-Time compilation explicitly *disabled* (`-Xint`). This provides a pure interpreter-versus-interpreter baseline between Java and Python.

5.2. Startup Overhead

To isolate the fixed cost of bringing each runtime up to an operational state before parsing begins, we evaluated two minimal startup variants:

- **The `light` variant (raw initialisation):** An empty entry point (`main`), capturing pure VM and process invocation overhead. In this baseline, Java pays a JVM initialisation penalty of $\sim$37 ms relative to Python (47.2 ms vs 10.5 ms).
- **The `heavy` variant (class and module loading):** A program that loads all classes and dependencies required by Pyttern without performing any parsing work. In this configuration, Java executes faster than Python (112.6 ms vs 156.0 ms), showing that Python's dynamic module import mechanism incurs substantial start-up overhead that exceeds the JVM's class-loading cost.

These two measurements establish the per-invocation fixed cost that any single-file command-line execution must amortise.

5.3. File-Size and Batch-Size Results

File-size workload To evaluate parsing scalability across varying lines of code (LOC), we extracted real-world Python source files from The Stack[3] (a large-scale repository of open-source code). For each target file size, files were sampled such that their line count fell within a tolerance margin of $\pm 5\%$ around the target LOC, retaining only syntactically valid files. As shown in Figure 4 (where the red box at the bottom-left of the right graph highlights the small-file region magnified in the left graph), for small files, the performance gap is modest (1.35$\times$ speedup). For typical file sizes ($\sim$100 LOC), Java parses $\sim$2.5$\times$ faster than Python. Crucially, the Java `-Xint` configuration closely tracks the Python curve across all file sizes (remaining within 10% of Python runtime), demonstrating that interpreted Java offers no structural advantage over Python.

Batch-size workload Evaluating directories of student submissions reveals that Java's performance advantage grows significantly with batch size, as shown in Figure 5. For a small batch of 50 files, Java is $\sim$7.5$\times$ faster than Python. On larger batches, the speedup approaches a factor of 20. In contrast, Java `-Xint` remains within 15% of Python across all batch sizes.

5.4. Discussion

The substantial speedup observed in batch mode is driven by two main factors:

1. **Amortisation of fixed costs:** In batch execution, the JVM is started once per directory, amortising initialisation overhead across all files.

[3] https://huggingface.co/datasets/bigcode/the-stack

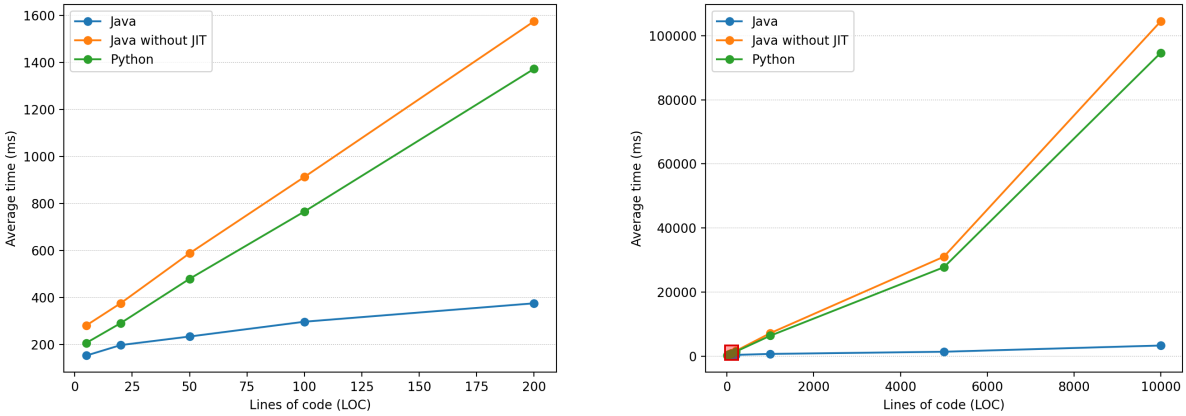


Figure 4: Average parsing execution time as a function of input size (LOC up to 200, left; up to 10,000, right). The red box in the right graph highlights the region zoomed in the left graph.

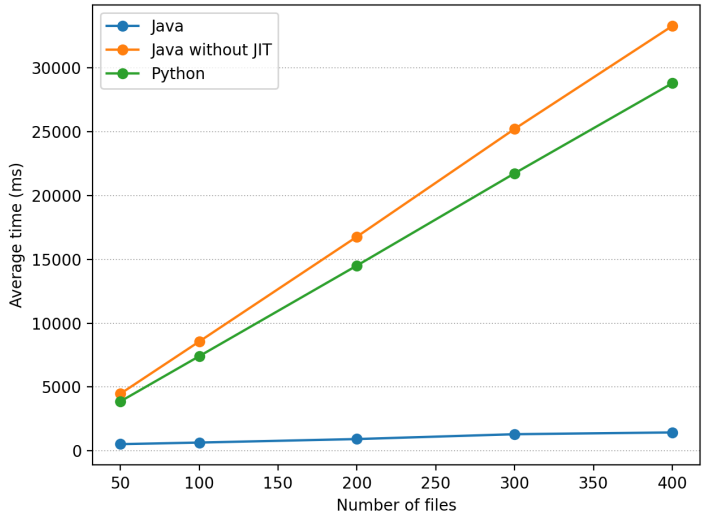


Figure 5: Average parsing execution time as a function of the number of input files.

2. **HotSpot adaptive compilation:** As the JVM executes the parser across multiple files in sequence, HotSpot's JIT compiler identifies hot paths and optimises them into native code [14, 15].

Because Java `-Xint` performs slightly worse than Python, we conclude that the performance gain stems entirely from HotSpot's adaptive compilation rather than intrinsic superiority of the Java ANTLR runtime library.

6. Matching Validation

This section evaluates Stage 2 of the port, comparing the full Java pipeline (parsing, PDA construction, and matching) against the original Python implementation.

6.1. Methodology

We evaluate matching performance at two levels of scope:

- **Macro-level:** External end-to-end timing using `hyperfine` [13], measuring parsing + PDA construction alone versus full execution.

- **Micro-level:** In-process function timing using `JMH` [16] for Java and `pyperf` [12] for Python to isolate internal matching performance.

The benchmarks use test cases from the Pyttern matcher test suite, which is partitioned into thematic categories targeting specific wildcard semantics: simple wildcards (`?`), named bindings (`?var`), list wildcards (`?*`), block indentation (`?:`), and multiple compound blocks (`multiple_body`, testing `?:*`). Each category pairs Python source files with matching and non-matching patterns that exercise various syntactic positions.

6.2. Results and Outlier Analysis

For the vast majority of pattern categories, matching runtime is negligible relative to parsing and PDA construction, confirming that Stage 1 parsing speedups translate directly to full pipeline improvements.

However, the `multiple_body` test category emerged as a prominent outlier (Figure 6). This category specifically stress-tests arbitrary block-nesting wildcards (`?:*`) within deeply nested control-flow structures (such as nested loops). In this category, matching dominates total runtime:

- **Micro-level matcher cost:** Python spends ∼750 ms in matching, whereas Java requires ∼8 ms.
- **Macro-level total cost:** Python end-to-end runtime reaches ∼1350 ms, compared to ∼230 ms for Java.

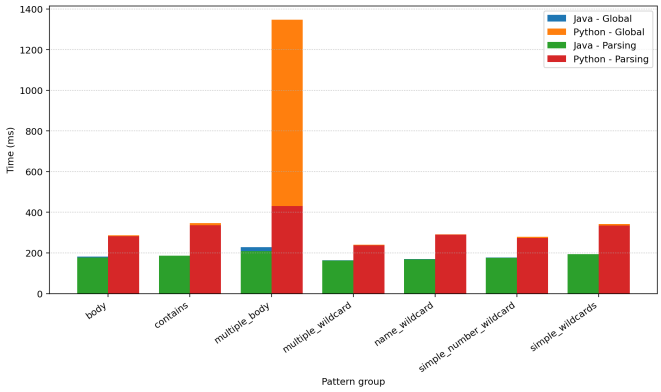


Figure 6: Average macro-level execution time per test category for the Java and Python implementations.

6.3. Discussion

The high matching cost in `multiple_body` stems from wildcards that match zero or more elements (`?:*` and `?*`). Each such wildcard forces the PDA to explore multiple branching paths during parse tree traversal.

Scaling with program size vs. pattern complexity These findings highlight a fundamental difference in how Pyttern's pipeline stages scale. As demonstrated in Section 5, parsing and PDA construction scale linearly ($O(N)$) with the input program size (LOC). Matching cost, in contrast, is largely decoupled from overall program size: it depends locally on the AST subtree being traversed and scales combinatorially ($O(k^d)$) with the number and nesting depth d of zero-or-more wildcards present in the query pattern (a comprehensive scaling analysis across file sizes up to 10,000 LOC is provided in [17]).

While Java significantly reduces constant-factor execution overhead via HotSpot compilation, it does not alter the underlying exponential state-space expansion. Addressing this combinatorial bottleneck requires targeted algorithmic matching optimisations.

7. Matching Optimisations

To eliminate the exponential matching bottleneck identified in Section 6, we developed three complementary optimisations targeting the `?:*` and `?*` wildcards.

7.1. PDA Optimisation: Skip Transition Correction

Analysis of the portion of the PDA generated for `?:*` revealed a structural bug in its transition rules. Upon entering a `?:*` wildcard, symbol B is pushed onto the stack, and a skip transition allows bypassing the body if zero blocks match.

However, when the automaton descends into a block, symbol I is pushed on top of B. The original skip transition lacked stack conditions, allowing the matcher to exit the wildcard even with I on top of B. This created redundant, identical exploration branches, effectively doubling match traces at each traversal level.

To fix this we constrained the skip transition to require symbol B at the top of the stack. This condition holds only upon initial entry and is violated after a descent (when I is on top), suppressing duplicate branches without altering matching semantics. The impact of this correction depends directly on the depth of nested `?:*` wildcards: on Pattern 4 (where `?:*` wildcards are separated by concrete statements), the speedup is a modest 12%. In contrast, on patterns with uninterrupted chains of nested `?:*` blocks, the fix prevents explosive branch duplication, yielding speedups of $48\times$ (Pattern 1), $2.4\times$ (Pattern 2), and $207\times$ (Pattern 3).

7.2. Parse Tree Pruning: Collapsing Consecutive Wildcards

Redundant sequences of wildcards (such as consecutive `?*` statements or nested `?:*` blocks) produce unnecessary non-deterministic choices in the PDA. We extended Pyttern's `Tree_Pruner` visitor with three rewriting rules that collapse consecutive, semantically equivalent wildcards at the parse tree level before PDA compilation:

1. Collapsing consecutive `?*` statement nodes inside a block.
2. Collapsing consecutive `?*` elements inside list/argument structures.
3. Collapsing directly nested `?:*` wildcard blocks.

By eliminating redundancy at the source, pruning prevents non-deterministic branching in the PDA entirely. On nested `?:*` wildcards, it achieves speedups of $130\times$ on Pattern 1 and $433\times$ on Pattern 3. On list `?*` wildcards, collapsing adjacent wildcards yields speedups of $198\times$ on Pattern 5 and $237\times$ on Pattern 8. On Patterns 6 and 7, where all wildcards are already isolated from each other by concrete elements, no rewriting rules apply and execution time is unchanged.

7.3. Matcher Optimisation: Counting Convergent Paths

During PDA traversal, multiple exploration paths frequently converge on identical configurations (defined strictly by the combination of automaton state, tree position, stack contents, and variable bindings). Standard memoization cannot be applied directly because skipping a previously seen configuration suppresses valid downstream matches.

To solve this we introduced a path-counting mechanism. When a new exploration path reaches a configuration with a signature (state, tree position, stack, and bindings) that is already waiting in the search queue, the matcher does not enqueue a duplicate configuration; instead, it increments a path counter associated with that pending configuration. Crucially, path merging occurs only when variable bindings are strictly identical; configurations with differing variable bindings represent distinct match contexts and continue to be evaluated separately. When a pending configuration is popped from the queue, its subtree is evaluated once, and any resulting matches are multiplied by its path count.

This path-sharing approach avoids redundant sub-graph traversals while preserving exact match counts and binding sets. It provides a consistent $\sim 4\times$ speedup across all nested block patterns in

`multiple_body` (reducing Pattern 1 from 690 ms to 170 ms and Pattern 3 from 2.9 s to 770 ms). In `multiple_wildcard`, it reduces Pattern 5 from 3.7 s to $< 200\,\mu$s ($> 18{,}000\times$) and Pattern 8 from 3.2 s to 29 ms ($110\times$).

7.4. Combined Evaluation and Interference

We evaluated the combined optimisations on two benchmark suites using `JMH`, numbered continuously from 1 to 8:

- **The `multiple_body` suite (Patterns 1–4):** Evaluates `?:*` wildcards inside nested loops. Patterns 1–3 contain uninterrupted chains of nested `?:*` blocks at increasing depth, whereas Pattern 4 separates wildcards with concrete statements.
- **The `multiple_wildcard` suite (Patterns 5–8):** Evaluates `?*` wildcards in statements and lists. Patterns 5 and 8 contain adjacent `?*` elements, while Pattern 7 contains consecutive statements.

Impact on global runtime On the `multiple_body` suite (Figure 7a), the combined optimisations reduced matcher execution time from 7.7 ms to 0.3 ms ($\sim$25$\times$ speedup). Total execution time dropped from 16 ms to 7 ms, fully aligning `multiple_body` with the other benchmark categories.

Interference analysis The three optimisations do not combine multiplicatively. Because parse tree pruning operates first, it removes redundant wildcards at the source, leaving fewer unpruned transitions for the PDA fix or path-counting matcher to optimize. Nevertheless, they cover complementary cases across both benchmark suites (Figure 7).

Irreducible combinatorial cost (Pattern 6) One benchmark pattern (`multiple_wildcard` Pattern 6, Figure 7b) resisted complete optimization, dropping from 4.3 s to 1.0 s. Pattern 6 consists of alternating `?*` wildcards separated by concrete integer elements (`[?*, 1, ?*, 1, ..., 1]`) matched against a list of 23 ones (`[1, 1, ..., 1]`). Because every wildcard is separated, the pruner cannot collapse them. Furthermore, the pattern generates an enormous number of legitimate, distinct integer partition matches that must all be enumerated. The matcher optimisation still accelerates exploration (from 4.3 s to 1.0 s) by sharing convergent sub-paths toward these distinct matches, but the residual cost is inherent: no redundancy-elimination technique can bypass enumerating thousands of distinct, valid matches.

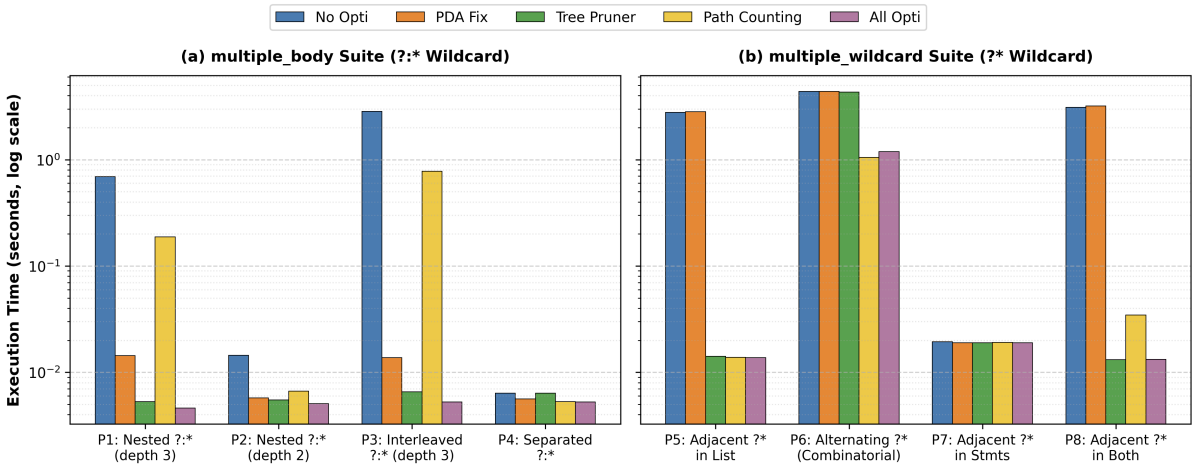


Figure 7: Execution time of optimisation variants on the `multiple_body` suite (Patterns 1–4, left) and `multiple_wildcard` suite (Patterns 5–8, right) on a log scale. The five bars per cluster represent: `No Opti` (baseline), `PDA Fix`, `Tree Pruner`, `Path Counting`, and `All Opti` (all combined).

8. Lessons Learned

Porting Pyttern from Python to Java provided practical insights into the engineering effort and architectural trade-offs of migrating AST-query runtimes.

Porting scope and engineering effort The full port was conducted over a six-month Master's thesis project [17]. The resulting Java codebase comprises approximately 3,500 lines of code (LOC) for the core parser, AST refinement visitor, PDA builder, and matching engine, supplemented by dedicated `JMH` microbenchmarking harnesses and an extensive regression test suite.

One-to-one translation vs. components redesigned The architectural elements differed significantly in how directly they could be translated:

- **One-to-one translation:** The ANTLR grammar specifications (`.g4`) for both Python and Pyttern were reused without modification. Similarly, the formal structure of the Pushdown Automaton—its state definitions, stack symbols (B and I), navigation instructions (`LEFT_CHILD`, `RIGHT_SIBLING`, `PARENT`), and core matching loop semantics—mapped directly 1-to-1 from the Python implementation.
- **Redesigned components:** In contrast, two major subsystems had to be fundamentally redesigned:
  1. *AST Visitor and Type Specialisation:* Python's dynamic duck-typing permitted a single polymorphic visitor to manipulate parse trees without explicit type declarations. In Java, navigating ANTLR's strongly-typed `ParserRuleContext` class hierarchy required designing explicit visitor bindings and type-safe dispatches, which constituted a major portion of the initial engineering effort.
  2. *Memory Architecture on Hot Paths:* In Python, search configuration snapshots were duplicated using shallow dictionary copies. In Java, a direct translation relying on object cloning created severe garbage collection pressure on the inner matching loop, heavily penalizing throughput. The configuration state was redesigned around persistent, immutable data structures with structural sharing, eliminating allocation overhead during transition exploration.

Runtime library discrepancies and error handling Although both runtimes execute identical grammar files, their error handling diverged: syntactically invalid input crashed the Python ANTLR runtime but was silently tolerated by the Java runtime. Implementing custom error listeners was necessary to ensure strict behavioral parity between implementations during automated benchmarking.

9. Future Work

This work opens several avenues for extending and refining Pyttern:

Large-scale multi-domain benchmarks While the profiling and validation studies in this paper relied on 50 student submissions from a single exercise, future work will scale empirical evaluation to large-scale, heterogeneous corpora of Python repositories (e.g., mined from GitHub or PyPI). Evaluating queries over thousands of real-world open-source packages will provide deeper insight into how Pyttern scales across diverse software architectures and programming idioms.

Declarative search and matching strategies Inspired by term-rewriting frameworks such as ELAN and Stratego, future extensions could integrate an orthogonal strategy language. Such an abstraction would allow query authors to declaratively guide non-deterministic exploration—such as toggling between eager first-match navigation (for rapid boolean linting checks) and exhaustive search (for comprehensive diagnostic reports), or specifying eager versus lazy matching over list wildcards.

JVM startup optimisation For single-file command-line invocations, JVM cold-start initialisation represents a notable portion of total execution time. Future work could explore deployment techniques such as Class Data Sharing (CDS) [18], Ahead-of-Time (AOT) class loading and profiling [15], or compiling the Java pipeline into a standalone binary via GraalVM Native Image.

Parser replacement In the optimised pipeline, parsing remains the primary contributor to total execution time. Investigating faster incremental parsing frameworks, such as Tree-sitter [19], represents a promising direction. Although replacing ANTLR would require porting Pyttern's extended grammars and rewriting parse-tree visitors, it could yield significant end-to-end performance gains.

10. Conclusion

In this tool and experience paper, we set out to make Pyttern fast enough to support high-throughput software maintenance, automated code linting, and analysis of large cohorts of student programs, without altering its matching semantics. Two identified bottlenecks were the parsing of input programs and the matching of patterns whose structure triggers combinatorial exploration.

The full Pyttern pipeline was reimplemented in Java in two stages. First, we replaced the Python ANTLR parser by one implemented in Java, yielding a substantial speedup already. Next, we reimplemented the pushdown automata construction and matching engine in Java, respecting the original matching semantics. Three algorithmic optimisations were developed on top of that to eliminate combinatorial non-determinism: (i) a correction in the pushdown automaton generation using stack-top guards to eliminate duplicate matches caused by an under-constrained skip transition on the `?:*` wildcard; (ii) an extension of Pyttern's tree pruner with three rewriting rules that collapse consecutive wildcards before the automaton is built; and (iii) a path-sharing matcher that merges exploration paths converging on identical configurations with path counting, preserving exact match counts and variable bindings.

For the parser, the port brought a speedup of about $2.5\times$ on individual files of typical size, and up to a factor of 20 in batch settings where JVM startup costs are amortised. A complementary configuration with HotSpot's JIT compiler disabled (`-Xint`) confirmed that adaptive compilation is the primary driver of this gain rather than runtime library maturity.

The three matching optimisations brought execution time down by several orders of magnitude on the most affected patterns (up to $207\times$ for the PDA fix, up to $433\times$ for tree pruning, and over four orders of magnitude for path counting). The optimisations act complementarily across pattern structures. Only one benchmark pattern (Pattern 6) resisted complete optimisation, retaining an execution time close to one second because it produces an enormous number of legitimate, distinct matches that must all be enumerated.

On the realistic workloads explored in this paper, the overarching goal is met: parsing is fast enough on batches of realistic size, and matching no longer dominates on the pattern groups that previously made it unusable. All artefacts, grammars, and benchmarking scripts are openly available to support replication and further research.

Data and Artifact Availability

The Java reimplementation of Pyttern, benchmark suites, and experimental evaluation scripts are publicly available on GitHub at https://github.com/Qlepine/Pyttern.

Acknowledgments & Declaration on Generative AI Use

This paper is a condensed version of the Master's thesis by the second author, Quentin Lépine [17], conducted under the supervision of the third and first authors, Kim Mens and Julien Liénard, respectively. During the preparation of this manuscript, the authors used Gemini 3.5 and 3.6 for grammar and spelling

correction, rephrasing, stylistic improvements, and text condensation. All AI-assisted modifications were extensively reviewed and iteratively refined by the authors, who take full responsibility for the final manuscript and its contents.

References

[1] J. Liénard, K. Mens, S. Nijssen, The Pyttern program query language, in: J. Edwards, R. Perera, T. Petricek (Eds.), Companion Proceedings of the 9th International Conference on the Art, Science, and Engineering of Programming (Programming 2025), volume 134 of *Open Access Series in Informatics (OASIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2025, pp. 23:1–23:15. URL: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.Programming.2025.23. doi:10.4230/OASIcs.Programming.2025.23.

[2] K. Mens, I. Michiels, R. Wuyts, Supporting software development through declaratively codified programming patterns, Expert Systems with Applications 23 (2002) 405–413. URL: https://doi.org/10.1016/S0957-4174(02)00076-3. doi:10.1016/S0957-4174(02)00076-3.

[3] M. Martin, B. Livshits, M. S. Lam, Finding application errors and security flaws using PQL: a program query language, in: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), ACM, 2005, pp. 365–383. doi:10.1145/1094811.1094840.

[4] E. Hajiyev, M. Verbaere, O. de Moor, codeQuest: Scalable source code queries with Datalog, in: ECOOP 2006 – Object-Oriented Programming, volume 4067 of *LNCS*, Springer, 2006, pp. 2–27. doi:10.1007/11785477_2.

[5] P. Klint, T. van der Storm, J. Vinju, RASCAL: A domain specific language for source code analysis and manipulation, in: 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2009), IEEE, 2009, pp. 168–177. doi:10.1109/SCAM.2009.28.

[6] D. Maier, K. T. Tekle, M. Kifer, D. S. Warren, Datalog: concepts, history, and outlook, in: M. Kifer, Y. A. Liu (Eds.), Declarative Logic Programming: Theory, Systems, and Applications, volume 20 of *ACM Books*, ACM / Morgan & Claypool, 2018, pp. 3–100. URL: https://doi.org/10.1145/3191315.3191317. doi:10.1145/3191315.3191317.

[7] N. Volanschi, C. Rinderknecht, Unparsed patterns: easy user-extensibility of program manipulation tools, in: Proceedings of the 2008 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2008), ACM, 2008, pp. 111–121. doi:10.1145/1328408.1328425.

[8] J. Harden, D. Kafura, A specification language for matching mistake patterns with feedback, in: Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE 2021), ACM, 2021, pp. 894–900. doi:10.1145/3408877.3432481.

[9] L. Gusukuma, A. C. Bart, D. Kafura, Pedal: An infrastructure for automated feedback systems, in: Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE 2020), ACM, 2020, pp. 100–106. doi:10.1145/3328778.3366913.

[10] Q. Colla, K. Mens, J. Liénard, A comparison of three program query languages to detect Python programming misconceptions, in: Companion Proceedings of the 9th International Conference on the Art, Science, and Engineering of Programming (Programming 2025), volume 134 of *OASIcs*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, pp. 21:1–21:15. doi:10.4230/OASIcs.Programming.2025.21.

[11] J. Liénard, Pyttern, https://github.com/JulienLie/Pyttern, 2026. Accessed: 2026-03-01.

[12] Python Software Foundation, pyperf: Toolkit to write, run and analyze benchmarks, https://github.com/psf/pyperf, 2026. Accessed: 2026-04-14.

[13] D. Peter, hyperfine: A command-line benchmarking tool, https://github.com/sharkdp/hyperfine, 2026. Accessed: 2026-03-08.

[14] Oracle, Java HotSpot virtual machine performance enhancements, https://docs.oracle.com/en/java/javase/11/vm/java-hotspot-virtual-machine-performance-enhancements.html, 2018. Java SE 11 documentation. Accessed: 2026-03-08.

[15] OpenJDK, JEP 515: Ahead-of-time method profiling, https://openjdk.org/jeps/515, 2025. Accessed: 2026-03-08.
[16] OpenJDK, JMH: Java microbenchmark harness, https://github.com/openjdk/jmh, 2026. Accessed: 2026-04-14.
[17] Q. Lépine, Reimplementing, Benchmarking and Optimising Pyttern in Java for Faster Pattern Matching, Master's thesis, Université catholique de Louvain (UCLouvain), 2026.
[18] OpenJDK, JEP 483: Ahead-of-time class loading and linking, https://openjdk.org/jeps/483, 2023. Accessed: 2026-03-08.
[19] M. Brunsfeld, other Tree-sitter contributors, Tree-sitter: An incremental parsing system for programming tools, https://tree-sitter.github.io/tree-sitter/, 2026. Accessed: 2026-09-09.