Visualising Python Pattern Matches at Scale with the Pyttern Visualiser

Julien Liénard[1,*], Berkay Murat[2] and Kim Mens[1]

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

Abstract

Automated assessment systems are widely deployed in large-scale programming courses to evaluate functional correctness. Providing timely targeted feedback on structural code quality, however, such as adherence to programming conventions, use of language idioms, and avoidance of anti-patterns, remains a challenge. Pyttern is a program query language designed to specify and detect code patterns and misconceptions using an intuitive, Python-like wildcard syntax. While Pyttern's matching engine currently supports compound queries composed by Boolean operators, its original visualisation tool was limited to inspecting isolated single-pattern and single-file matches for debugging purposes only. Because of this limited visualisation support, educators struggle using Pyttern to efficiently evaluate structural queries over larger student cohorts.

This paper introduces the **Pyttern Visualiser**, an interactive web application designed to support pattern-based structural analysis and feedback with Pyttern at scale. The visualiser offers a Compound View that enables teachers to import hierarchical pattern trees composed with boolean operators, upload and execute batches of student submissions, interactively filter submissions by similar sub-pattern outcomes, and export the results in CSV format. Furthermore, it seamlessly combines these cohort-level summaries with fine-grained analysis through deep-dive views on individual matches. We validated the visualisation tool through automated test suites (123 unit and integration tests) and an empirical usability evaluation with 15 participants. The tool achieved an average SUS score of 79, demonstrating its effectiveness, clarity, and readiness for real-world educational workflows.

Keywords

Program Query Language, Pattern Matching, Automated Grading, Computer Science Education, Software Visualisation, Static Analysis, React, Usability Evaluation

1. Introduction

The rapid growth of student enrollment in computer science courses has made automated assessment systems and autograders indispensable in modern computing education [1]. While existing autograding platforms reliably verify functional correctness using black-box unit tests or input/output evaluation, they often fall short in assessing structural software quality [2]. Ensuring that a program passes test cases is not sufficient, however. Students must also learn to adhere to language idioms and programming conventions, avoid harmful anti-patterns [3], follow object-oriented design principles, and employ appropriate algorithmic constructs. Providing timely, formative, and targeted feedback on such structural code regularities at scale remains an important pedagogical challenge.

To facilitate structural static analysis, Program Query Languages (PQLs) allow educators and software engineers to specify and detect specific syntactic and structural patterns in source code [4]. Among these, *Pyttern* [5] was developed as a developer-friendly PQL for Python. Unlike classical PQLs that require mastering declarative logic engines (e.g., Datalog [6]) or navigating Abstract Syntax Tree APIs via XPath or visitor objects, Pyttern [5] enables queries to be expressed directly in concrete Python-like syntax augmented with intuitive wildcard abstractions.

The 25th Belgium-Netherlands Software Evolution Workshop, 26–27 November 2026, Madrid, Spain
*Corresponding author.
julien.lienard@uclouvain.be (J. Liénard); berkay.murat@student.uclouvain.be (B. Murat); 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).

Although Pyttern's matching engine is capable of evaluating compound queries, composed of more primitive queries through Boolean operators, across multiple source files, via its command-line interface, its graphical tooling is still lagging behind. Because the original Pyttern visualiser (which we will refer to as the 'Legacy Visualiser' in the remainder of this paper) was conceived as a low-level debugging instrument for language designers, its interface is limited to visualising how one isolated pattern matches against one isolated source code file. Consequently, an instructor seeking to assess whether hundreds of student submissions satisfy a more complex pattern composed of multiple smaller ones (e.g., checking a more complex design pattern consisting of several smaller structural regularities) is forced to execute hundreds of repetitive manual runs and mentally reconcile the disparate outputs. This interface limitation creates an operational bottleneck that prevents educators from utilising Pyttern for cohort-level pedagogical assessment on compound patterns.

Real-world teaching scenarios demand multi-file batch execution, composition of multiple sub-patterns via logical operators (AND, OR, NOT), aggregate visualisation of cohort success rates, and interactive filtering to isolate specific student misconceptions or specific sub-patterns that several students seem to struggle with. Furthermore, when an unexpected mismatch occurs, instructors need an effortless transition from high-level cohort summaries to fine-grained, step-by-step match execution for individual students or sub-patterns without having to re-upload files or lose their session context.

To bridge this gap between detailed pattern matching capabilities and classroom-wide workflows, the new Pyttern Visualiser proposed in this work pursues the following four objectives:

Compound Pattern and Multi-File Visualisation Design and implement a dedicated *Compound View* supporting hierarchical Boolean pattern trees (AND, OR, NOT) and batch uploading of student submissions.

Interactive Exploration and Filtering Provide interactive filtering mechanisms that allow educators to slice cohort results based on arbitrary sub-pattern matching criteria and export structured CSV grade sheets.

Seamless Matcher Deep-Dive Integration Establish a bidirectional bridge between the high-level Compound View and the low-level Matcher View (i.e., the legacy visualiser), enabling one-click drill-down into detailed automaton execution for any pattern–submission pair.

Implementation and Validation Implement the frontend using a robust, modular, layered Redux architecture, validate the codebase with comprehensive automated testing, and empirically evaluate usability with end users.

To achieve these objectives we adopted an iterative, user-centered design methodology [7]. The visualiser is conceived as a responsive React Single Page Application (SPA) backed by the Redux Toolkit for state management, interacting asynchronously with the Pyttern backend matching engine. We validate its implementation through a comprehensive test suite of 123 automated unit and integration tests, followed by an empirical usability study involving 15 participants evaluating realistic pedagogical grading tasks using the System Usability Scale (SUS) [8, 9].

The remainder of this paper is structured as follows. Section 2 discusses related work. Section 3 introduces the Pyttern language and its legacy visualiser. Section 4 revisits our problem statement by presenting a motivating educational scenario. Section 5 details the design and interactive features of the novel Pyttern Visualiser. Section 6 discusses the visualiser's frontend software architecture and state management. Section 7 presents our validation through automated tests and usability study results. Finally, Section 8 outlines future work and concludes the paper.

2. Related Work

Program Query Languages and Code Analysis A wide range of Program Query Languages (PQLs) and static analysis tools have been developed to detect patterns, bugs, and design violations in source code [4]. Foundational work in declarative and logic meta-programming (LMP) was pioneered by SOUL [10], which demonstrated how structural regularities, design patterns, and architectural constraints could be codified and queried as logic predicates in symbiotic reflection with object-oriented

programs. While logic-based query languages provide powerful declarative reasoning, authoring queries as formal logic rules imposes a steeper learning curve and significant cognitive overhead for non-expert developers such as educators.

Alternative static analysis tools, like PMD [11] and Checkstyle [12], rely on XPath queries or Abstract Syntax Tree (AST) visitors to identify structural flaws. While effective for predefined linting rules, defining custom queries similarly requires extensive familiarity with underlying parser grammars and tree representations. Modern AST-based query engines such as Semgrep [13] utilise concrete-syntax patterns similar to Pyttern. However, Semgrep's query specifications are primarily oriented toward security auditing and rule-based CI/CD pipelines rather than interactive educational exploration.

Pyttern [5] was introduced specifically to combine the expressiveness of concrete syntax with wildcard matching tailored for Python, and has been adapted to Java via Jattern [14, 15] and optimised for more performant matching [16]. In an empirical comparison of PQLs for detecting Python misconceptions, Colla et al. [3] demonstrated that concrete-syntax PQLs like Pyttern significantly reduce the cognitive barrier for educators as compared to syntax tree-based and logic-based formalisms (such as those stemming from the LMP tradition). Our work builds upon this foundation by providing the interactive visualisation and cohort-level tooling necessary to deploy Pyttern in classroom environments.

Automated Grading and Structural Feedback Automated grading platforms (such as INGInious, Dodona, Web-CAT, and Gradescope) have transformed computing education by enabling scalable grading of large student cohorts [1]. However, these platforms traditionally emphasise dynamic execution and unit testing, verifying *whether* code produces expected outputs or satisfies given tests rather than *how* the code is structured [2].

Educational research has shown that automated feedback is most effective when it provides actionable guidance on structural idioms, programming conventions, anti-patterns, etc. [17]. While static analysis tools have been embedded into autograding pipelines, instructors frequently lack interactive mechanisms to specify, verify, and provide feedback on complex structural patterns over batch submissions of larger student cohorts. Our new Pyttern Visualiser aims to address exactly this need.

Interactive Software and Grammar Visualisers Visual debuggers and interactive query builders have long played a vital role in making complex formalisms accessible. For instance, interactive regular expression tools like *regex101* allow developers to compose patterns while visualising token matches, automaton backtracking, and capture groups in real time. In the compiler domain, parser visualisers display parse trees and automaton transitions to help developers debug grammars.

In the program querying domain, early logic-based and AST querying tools either remained headless command-line utilities or provided single-file IDE integrations. Our Pyttern Visualiser advances the state of the art by integrating macro-level cohort exploration (hierarchical Boolean pattern trees, matching of batches of submissions, and filtering based on sub-pattern matching) with micro-level automaton stepping in a unified, interactive web environment.

3. Pyttern and its Legacy Visualiser

Overview of the Pyttern Language *Pyttern* is an open-source program query language designed to detect syntactic patterns and anti-patterns in source code [5]. Unlike query systems based on AST visitors, XPath expressions, or logic programming, Pyttern adheres to a *concrete syntax* paradigm: queries are expressed using standard Python syntax augmented with pattern wildcards.

Listing 1 demonstrates a simple Pyttern pattern designed to detect inheritance relationships. The query specifies a parent class `?Parent` and a child class `?Child` extending `?Parent`, separated by arbitrary intermediate contents (`?*`). When evaluated against the student code in Listing 2, the matcher binds `?Parent` to `Vehicle` and `?Child` to `Car`, yielding a successful match.

Pyttern offers a variety of different wildcards. The simple wildcard `?` matches single AST nodes, identifiers or literals. Named wildcards (e.g., `?var`, `?func`) enforce consistent matches of the wildcard

across the query scope. The list wildcard `?*` matches lists of AST nodes, and `?{n,m}` can be used to match between n and m nodes in a list. Subtree wildcards like `?:`, `?:*` or `?<...>` match statement blocks, nested blocks, or specific expressions within a block, respectively.

```
class ?Parent:
    ?*
class ?Child(?Parent):
    ?*
```

Listing 1: Pyttern Pattern (`has_parent_child.pyt`)

```
class Vehicle:
    def __init__(self, brand):
        self.brand = brand

class Car(Vehicle):
    def __init__(self, brand, doors):
        super().__init__(brand)
        self.doors = doors
```

Listing 2: Matching Python Code (`vehicle.py`)

Compound Patterns When assessing student assignments, an educator often wants to express more complex patterns that consist of several subpatterns combined through logical operators (and, or, not). The version of Pyttern that was used in this paper [5] does not support such logical operators at the language level, but does allow multiple individual patterns to be composed into more complex ones at the file level. The idea is simple: *primitive patterns* (without logical operators) are stored in separate `.pyt` files. These can be composed into *compound patterns* by placing them in *operator folders* named AND, OR, or NOT, where the name of the folder determines how the results of the patterns contained in those folders should be combined. Such compound patterns can then further be composed into even more complex compound patterns, as illustrated in Figure 1. A compound query is satisfied if and only if the root operator evaluates to `true` according to standard Boolean semantics over its child subtrees.

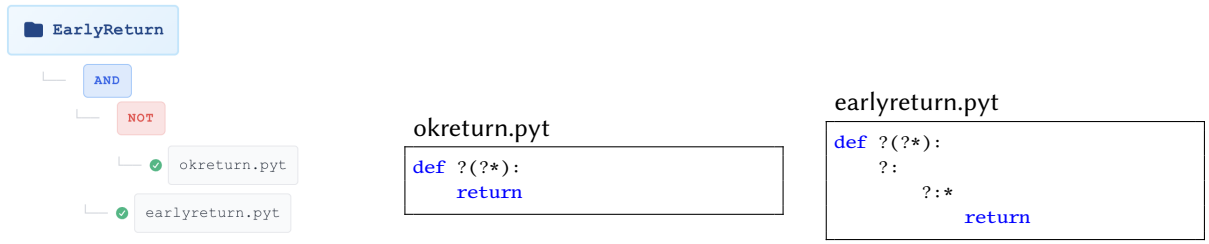


Figure 1: Pyttern's folder structure for using Booleon operators to combine primitive patterns into more complex compound pattern.

The Legacy Pyttern Visualiser The original Pyttern visualiser, shown in Figure 2, was mainly conceived as a tool to help the Pyttern language designers or expert users understand the matching process at a fairly detailed level. Furthermore, it focused only on visualising how a single primitive `.pyt` pattern file on the left matches a single `.py` source code file on the right.

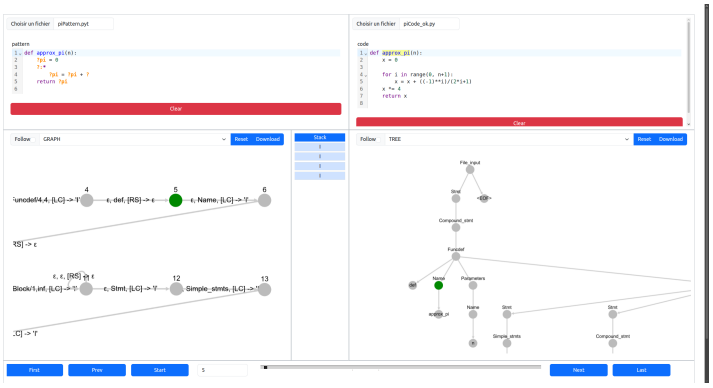


Figure 2: The Legacy Pyttern Visualiser (single-file Matcher View).

Clicking the "Start" button initiates the matching process, rendering:

1. Syntax Trees for both pattern and target code.
2. A Pushdown Automaton (PDA), the underlying formalism upon which Pyttern relies, specifying how the Pyttern pattern navigates the source code to be matched.
3. An interactive step-by-step debugger showing automaton transition states, token consumption, and PDA stack operations.
4. Synchronized source code highlighting corresponding to the current syntax tree nodes being visited by the step-by-step debugger.

While this fine-grained visualisation is invaluable for understanding the parse tree navigation and automaton transitions within the Pyttern runtime engine, it is less suited for high-level understanding of (mis)matches during grading assignments or for providing educational feedback. It also lacks mechanisms for dealing with compound patterns, batch-processing of multiple student submissions, and summarising cohort results.

4. Motivating Scenario

To illustrate the practical limitations of the Legacy Pyttern Visualiser, consider an instructor teaching an intermediate Object-Oriented Programming (OOP) module in Python. Students are assigned to implement a vehicle hierarchy featuring a base class `Vehicle` and one or more specialised subclasses (e.g., `Car`, `Truck`). To assess whether the students' submissions comply with sound object-oriented practices beyond functional tests, the instructor establishes a compound pattern named `ProperInheritance`. This pattern is composed of five more primitive patterns, summarised in Table 1. The overall Boolean compound pattern can be formulated as:

$$\begin{aligned} \texttt{has_parent_child} \wedge \texttt{has_super_init} \wedge \neg\,(\,\texttt{has_init_return}\,) \\ \wedge\,(\,\texttt{has_method_return} \vee \texttt{has_early_return}\,) \end{aligned} \tag{1}$$

Primitive Pattern	Structural Regularity	Role in Compound Rule
`has_parent_child.pyt`	Base class and derived subclass definition	Required (AND)
`has_super_init.pyt`	Subclass constructor invokes `super().__init__()`	Required (AND)
`has_init_return.pyt`	`__init__` constructor explicitly returns a value	Anti-Pattern (NOT)
`has_method_return.pyt`	Method with a standard return statement	Alternative 1 (OR)
`has_early_return.pyt`	Method with an early return inside a conditional	Alternative 2 (OR)

Table 1
Primitive Pyttern patterns composing the `ProperInheritance` compound pattern.

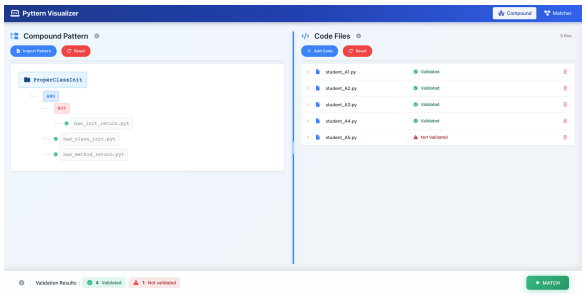
Now suppose the course instructor receives submissions from 40 different students and wishes to identify which students satisfy the entire compound pattern, which students introduced the constructor return anti-pattern, and which students failed to invoke parent constructors. Attempting to execute this assessment scenario using the legacy Pyttern Visualiser reveals severe usability barriers. First, because the legacy visualiser supports matching only a single pattern and a single code file per run, evaluating 40 student submissions against the 5 primitive patterns would require $40 \times 5 = 200$ distinct, pairwise upload-and-match operations, causing a huge manual overhead. Second, the instructor must manually record the individual Boolean results for each submission on an external spreadsheet, and mentally evaluate the compound formula (Equation 1), to calculate final grades or provide feedback. Third, there is no aggregate overview to observe cohort-wide distributions, making it impossible to quickly determine common misconceptions among student submissions (e.g., discovering that 50% of the class forgot to call `super().__init__()`). Finally, if an unexpected failure occurs in a student's code, debugging requires completely resetting the view, re-uploading the specific pattern and source file into the Matcher, and restarting the analysis.

To overcome these limitations and turn Pyttern into an effective pedagogical tool, we need a novel Pyttern Visualiser that fulfills the following key operational requirements:

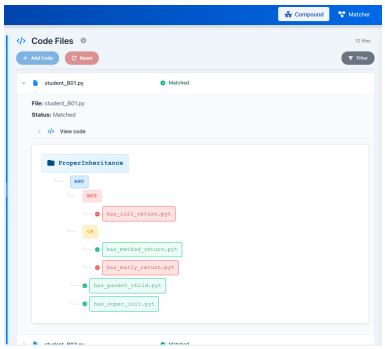
1. Allow instructors to *import compound pattern directory structures* in a single drag-and-drop action, validating tree syntax and operator validity automatically.
2. Support *batch uploading and execution* of submissions of entire student cohorts, dispatching parallel matching queries against the backend engine.
3. Present an *intuitive dashboard* displaying overall pass/fail status per submission, alongside clickable cards showing matching status for each pattern file.
4. Enable instructors to *filter* student submissions dynamically based on arbitrary combinations of sub-pattern outcomes (e.g., isolate all files matching `has_parent_child` but failing `has_super_init`).
5. Enable instantaneous navigation from any leaf node in a student's result card to the *detailed Matcher View*, pre-populating the relevant code and pattern without data loss.

5. The Extended Pyttern Visualiser

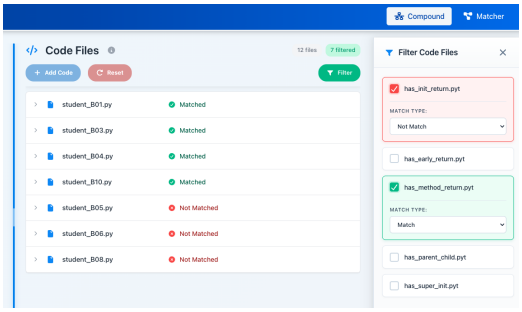
The extended Pyttern Visualiser transforms Pyttern from an isolated debugging tool into a large-scale interactive analysis system. The visualiser is conceived as a decoupled client-server web application. The backend exposes the Pyttern runtime engine via RESTful HTTP endpoints for syntactic validation, single-file automaton matching, and batch execution of compound patterns. The frontend is built as a web application using React [18] and Redux Toolkit [19], managing caching, filtering, and transitions between its different views.



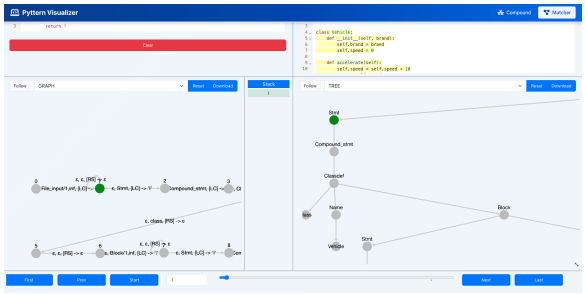
(a) Compound View with pattern tree and cohort files.



(b) Submission cards with subpattern matches.



(c) Interactive multi-criteria filtering drawer.



(d) Matcher View with PDA state and AST steps.

Figure 3: The extended Pyttern Visualiser: (a) cohort-level compound overview, (b) expanded submission cards, (c) in-browser filtering drawer, and (d) integrated single-pattern PDA matcher.

The Compound View (Figure 3a) serves as the primary operational workspace for educators. Its interface is organised as a responsive split-screen layout containing two core panels:

Hierarchical Pattern Tree (left) When the user imports a compound pattern folder, the client parses the directory recursively, rendering operator nodes (AND, OR, NOT) and clickable leaf query nodes (`.pyt` files). Then, the system displays syntax validation badges after backend validation.

Submission Cards (right) Each individual student submission appears as an expandable card with status badges *Not Validated* (not parsable, e.g. due to a syntax error), *Validated* (parsable but not yet matched), *Matched* (matches the compound pattern on the left), *Not Matched* (doesn't match the pattern). Expanding the card for an individual submission (Figure 3b) displays the full compound pattern tree annotated specifically for that student's code, including its match status for each of the subpatterns of the compound pattern.

The bottom toolbar coordinates the lifecycle of batch analysis (bottom of Figure 3a). Before matching, the system merely verifies that the compound pattern tree and all uploaded student source files pass syntax checks (*Validated* or not). Clicking "Match" triggers asynchronous requests to the backend to match each idividual submission. During this matching process, individual submission cards are updated dynamically with live progress indicators, culminating in aggregate cohort summary statistics (e.g., "4 Matched, 8 Not Matched").

The Filtering drawer (Figure 3c) In large classes, inspecting hundreds of student submissions is tedious. The extended visualiser offers an in-browser filtering engine accessible as a slide-out drawer where instructors can configure multi-criteria filter rules (e.g., `has_init_return == false` AND `has_method_return == true`). Filtering happens instantly on the client side over cached Redux state without network latency. Furthermore, a one-click CSV export generates structured sheets recording filenames, overall outcomes of compound matches, and individual match statuses for every leaf pattern.

Integration with the Matcher View. A key innovation of the extended visualiser is its seamless integration between high-level analysis and fine-grained automaton inspection. Every leaf node in an expanded student card acts as an interactive link that, when clicked, transitions the visualiser into the *Matcher View* (Figure 3d), pre-loading the specific student code and pattern query. In that detailed view, instructors can step through PDA transitions, inspect AST node highlighting, and observe stack operations to diagnose match failures, while a navigation header preserves complete compound session state for instant return.

6. Implementation and Architecture

Modern web frontends for complex tasks require clean architectural separation to prevent component coupling [20]. The Pyttern Visualiser codebase is structured as a *feature-based modular architecture* [19]. Rather than organising files by technical role, the project encapsulates all logic into self-contained feature packages: `features/compound/`, `features/matcher/`, and `common/`. Cross-feature dependencies are strictly avoided; communication occurs exclusively through the global Redux store and well-defined navigation parameters.

Within each feature module, data flows through a *six-layer data flow architecture* organised into a Redux state loop and a backend communication pipeline (Figure 4):

UI Components Pure React functional components rendering the interface, capturing user events, and dispatching actions.

Async Thunks Action orchestrators managing asynchronous workflows, sequencing validation checks and batch matching calls.

Redux Slices Pure reducer functions governing immutable state mutations using Redux Toolkit's built-in Immer integration.

API Services Feature-level modules performing domain data transformation, payload structuring, and mapping backend responses.

API Client A centralised Axios HTTP layer communicating with backend REST endpoints with standardised error handling.

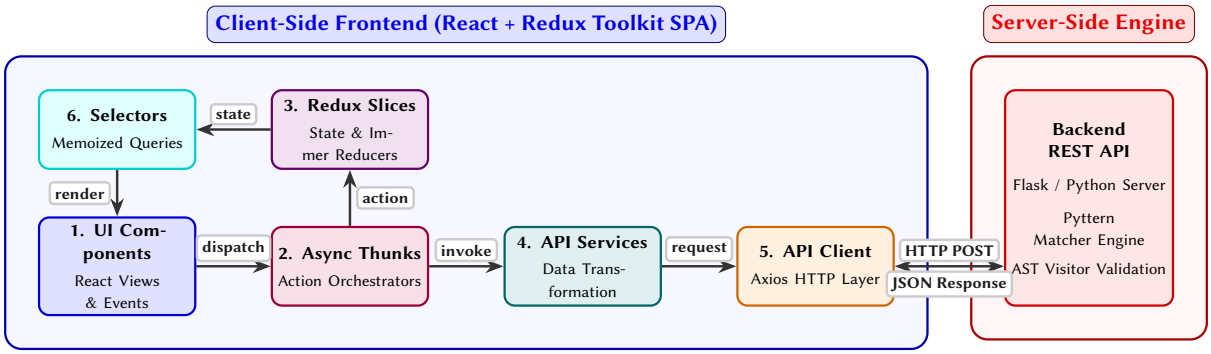


Figure 4: Architecture and six-layer internal frontend data flow of the extended Pyttern Visualiser.

Selectors Memoised query functions constructed with `reselect`, deriving optimised view models to drive component re-rendering.

Maintaining coherent state during deep-dive match inspections was a key technical objective. When a user navigates from the Compound View to inspect a specific leaf pattern in the Matcher View, the active compound session state, including all loaded files, pattern hierarchies, match statusses, and filter settings, is retained in the global Redux store. To prevent accidental data loss during such detailed matching sessions, the application implements navigation guards: attempting to leave an active compound session or reload the page prompts an explicit confirmation modal, protecting users from losing batch execution results.

7. Evaluation

7.1. Automated Software Validation

To ensure the reliability, maintainability, and correctness of the extended Pyttern Visualiser, we constructed an automated test suite using `Vitest` and `React Testing Library`. The test suite comprises **123 automated tests** organised across four architectural categories: **Tree Parsing and Utilities** (42 tests) validates directory parsing, Boolean operator extraction, path normalisers, and CSV serialisation; **Redux Slices and State Transitions** (38 tests) exercises state immutability, action dispatching, error capture, and cache invalidation; **Memoised Selectors and Filtering** (21 tests) verifies complex multi-criteria filter predicates, edge-case evaluations (e.g., handling missing or invalid files), and selector memoisation efficiency; and **Asynchronous Thunk Integration** (22 tests) simulates end-to-end network pipelines, batch request sequencing, error recovery, and backend synchronisation. All 123 tests execute in under 4 seconds in local environments and CI pipelines, achieving comprehensive branch coverage across core business logic.

7.2. Usability Evaluation Setup

To evaluate the pedagogical usability and interface ergonomics of the extended Pyttern Visualiser, we conducted an empirical user study with **15 participants**. All participants were Master's-level Computer Science students enrolled in a *Programming Paradigms* course. Participants possessed university-level programming knowledge and familiarity with software structure and static analysis principles.

Each evaluation session lasted approximately 45 minutes and proceeded in four structured phases. First, in a **Tutorial and Orientation** phase, participants received a 10-minute briefing introducing Pyttern's wildcard syntax, compound Boolean folders, and the visualiser workflow. Second, in **Task A (Basic Compound Assessment)**, participants evaluated 6 student submissions against a basic compound rule (`ProperClassInit`) combining 3 sub-patterns checking constructor definitions and method return statements. Third, in **Task B (Complex Cohort Assessment)**, participants evaluated 12 student submissions against the comprehensive `ProperInheritance` rule (Equation 1) comprising 5 sub-patterns across nested `AND`, `OR`, and `NOT` operators, requiring them to batch-match submissions,

use the filtering interface to identify students who missed `super().__init__()`, inspect failing cases in the Matcher View, and export the resulting match sheet. Finally, in a **Post-Test Survey**, participants completed a standard System Usability Scale (SUS) questionnaire [8] along with more specific questions and open-ended feedback.

7.3. Usability Results and Analysis

System Usability Scale (SUS) Score The system achieved an average overall SUS score of **79.0**, with a median of 80.0 and a standard deviation of 15.5, across all 15 participants. According to empirical SUS benchmark literature [9, 21], a score of 79 places the system above the 'Average' benchmark threshold (68.0), qualifying the user experience as 'Good' to 'Excellent' and indicating high overall usability. The per-item view in Figure 5 shows where this satisfaction comes from. The strongest items were Q6 on consistency (92%) and Q5 on integration (85%), which indicates that participants saw the interface as coherent. This is a meaningful result because the Compound page was added on top of the existing Matcher view rather than replacing it. The lowest item was Q7 on how quickly most people would learn the tool (72%). While this score is still positive, it does point to the learning curve of the compound pattern model as the dimension most worth improving.



Figure 5: SUS Agreement Percentages per item of the questionnaire ($N = 15$)

Workflow-Specific Evaluation In addition to the standard SUS instrument, participants evaluated four domain-specific questions rated on a 5-point Likert scale. The four workflow-specific questions all received average ratings well above 4 on the five-point scale, with a combined average of 4.40. The distribution of answers for each question is shown in Figure 6. The strongest result was Q13, which shows that participants could reliably distinguish between matched, not matched, and not validated files. The filter feature (Q12) was widely seen as efficient for narrowing down the visible files. Q14 asked whether the tool would be useful in real teaching scenarios. Even participants who reported some difficulties in the open feedback still rated the tool as useful for teaching, which suggests that the

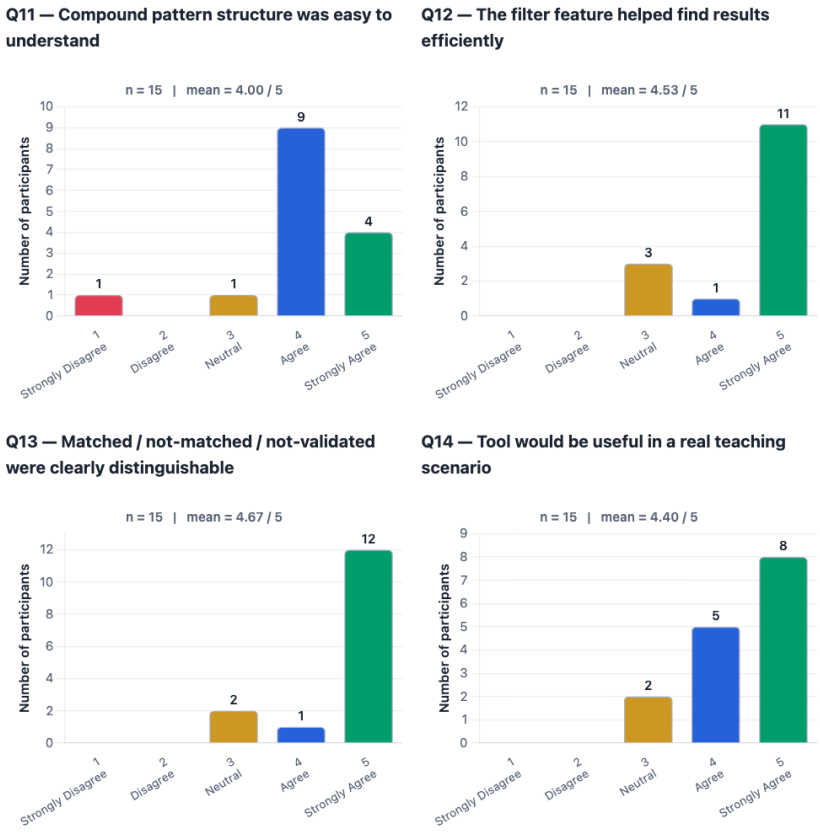


Figure 6: Usability evaluation: workflow ratings (Q11–Q14).

limitations they raised are perceived as polish items rather than fundamental barriers. Question Q11 on the ease of understanding of the Compound Pattern View, had the widest spread of the four questions, but still obtained agreement of 13 of the 15 participants.

Qualitative Observations Qualitative feedback collected during the evaluation highlighted three primary user strengths. First, participants emphasised that batch matching and automated evaluation of compound patterns eliminated the repetitive administrative work that would otherwise be required when assessing such patterns manually. Second, the clickable submission cards allowed users to immediately identify which subpattern(s) of a compound pattern caused a rejection, without having to inspect source files line by line. Third, participants praised the seamless round-trip navigation between high-level filtering and detailed matching. Suggested areas for future enhancement included adding in-browser pattern creation tools and expanding the filter query to support nested logical conditions.

7.4. Threats to Validity

Several threats may have affected the validity of our findings. Regarding *construct validity*, our study assessed subjective user perception through SUS and Likert questionnaires rather than measuring objective grading throughput or error-detection rates against a baseline control. Additionally, participants inspected pre-curated submissions with pre-authored pattern trees, whereas real educational workflows require educators to author patterns from scratch and inspect uncurated code with syntax anomalies or partial solutions. Regarding *internal validity*, the fixed task sequence from basic constructor checks to nested inheritance rules introduced potential learning effects across tasks, while the tutorial briefing and moderator presence may have mitigated initial onboarding hurdles, such as understanding folder-based Boolean operators. Finally, regarding *external validity*, our 15 participants were master students performing simulated tasks rather than actual teachers managing real courses. Furthermore, evaluating small batches of up to 12 submissions does not expose potential browser rendering overhead, memory

pressure, or visual clutter that may emerge with cohorts of several hundred of students, nor does it capture integration within institutional learning management systems.

8. Conclusion and Future Work

Providing timely, targeted feedback on structural code quality, coding and design idioms is essential in computer science education, yet automated grading platforms predominantly focus on functional testing. In this paper, we presented the extended **Pyttern Visualiser**, an interactive web application designed to support pattern-based structural analysis across student cohorts. By introducing a hierarchical Compound View, batch execution pipelines, interactive multi-criteria filtering, and seamless transitions to fine-grained PDA automaton stepping, the tool bridges the gap between low-level pattern matching and classroom assessment workflows. The application was developed following a modular six-layer Redux architecture, verified through 123 automated tests, and evaluated in an empirical usability study with 15 participants, achieving an average System Usability Scale (SUS) score of 79.0 which indicates good usability.

Several promising avenues for future research and engineering emerge from this work. First, to eliminate the need for manual filesystem editing of pattern files, we plan to develop an in-browser graphical pattern builder with interactive wildcard matching, live syntax validation, and template scaffolding. Second, integrating the visualiser directly into automated assessment platforms such as INGInious will enable instructors to author compound patterns that deliver immediate, personalized structural feedback upon code submission. Finally, with the recent generalisation of Pyttern to Java via Jattern [14, 15] and ongoing performance optimizations [16], we envision extending the tool to support multi-language compound querying within a unified interface, as well as leveraging match vectors to automatically cluster submissions and discover recurring misconceptions across cohorts.

Acknowledgments & Declaration on Generative AI Use

This paper is a condensed version of the Master's thesis by the second author, Berkay Murat [22], 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] G. Haldeman, A. Tjang, M. Babeş-Vroman, S. Bartos, J. Shah, D. Yucht, T. D. Nguyen, Providing meaningful feedback for autograding of programming assignments, in: Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18), 2018, pp. 278–283. doi:10.1145/3159450.3159502.

[2] H. Keuning, Automated feedback for learning code refactoring, Ph.D. thesis, Open Universiteit, Heerlen, The Netherlands, 2020. URL: https://research.ou.nl/en/publications/automated-feedback-for-learning-code-refactoring.

[3] Q. Colla, K. Mens, J. Liénard, A comparison of three program query languages to detect python programming misconceptions, in: OpenAccess Series in Informatics (OASIcs), volume 134, 2025, pp. 21:1–21:12. doi:10.4230/OASIcs.Programming.2025.21.

[4] S. Paul, A. Prakash, A framework for source code search using program patterns, IEEE Transactions on Software Engineering 20 (1994) 463–475. doi:10.1109/32.295894.

[5] J. Liénard, K. Mens, S. Nijssen, Pyttern: a python-based program query language, in: Proceedings of the 23rd Belgium-Netherlands Software Evolution Workshop (BENEVOL 2024), volume 3941 of

CEUR Workshop Proceedings, CEUR-WS.org, 2024, pp. 88–96. URL: https://ceur-ws.org/Vol-3941/BENEVOL2024_TECH_paper10.pdf.

[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] J. M. Carroll, Human-computer interaction: Psychology as a science of design, International Journal of Human-Computer Studies 46 (1997) 501–522. doi:10.1006/ijhc.1996.0101.

[8] J. Brooke, SUS: A 'quick and dirty' usability scale, Usability evaluation in industry 189 (1996) 4–7.

[9] A. Bangor, P. T. Kortum, J. T. Miller, An empirical evaluation of the system usability scale, International Journal of Human-Computer Interaction 24 (2008) 574–594. doi:10.1080/10447310802205776.

[10] R. Wuyts, K. Mens, Codifying structural regularities of object-oriented programs, in: Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 37), 2001, pp. 88–99. doi:10.1109/TOOLS.2001.911760.

[11] PMD Development Team, PMD source code analyzer documentation, 2026. URL: https://docs.pmd-code.org/latest/index.html.

[12] Checkstyle Development Team, Checkstyle, 2026. URL: https://checkstyle.sourceforge.io/.

[13] Semgrep Inc., Semgrep documentation, 2026. URL: https://semgrep.dev/docs/.

[14] N. Dedoyard, Jattern: A program query language for Java, Master's thesis, Université catholique de Louvain, 2025. URL: https://hdl.handle.net/2078.2/43874.

[15] L. Leurquin, Generalising Pyttern: porting a Python-based program query language to Java, Master's thesis, Université catholique de Louvain, 2026.

[16] Q. Lépine, Reimplementing Pyttern in Java for Faster Pattern Matching, Master's thesis, Université catholique de Louvain, 2026.

[17] S. Narciss, Feedback strategies for interactive learning tasks, Handbook of Research on Educational Communications and Technology 3 (2008) 125–144.

[18] M. Madsen, O. Lhoták, F. Tip, A semantics for the essence of React, in: Proceedings of the 34th European Conference on Object-Oriented Programming (ECOOP 2020), volume 166 of *LIPIcs*, 2020, pp. 12:1–12:26. doi:10.4230/LIPIcs.ECOOP.2020.12.

[19] D. Pronina, I. S. Kyrychenko, Comparison of Redux and React Hooks methods in terms of performance, in: Proceedings of the 6th International Conference on Computational Linguistics and Intelligent Systems (COLINS 2022), volume 3171 of *CEUR Workshop Proceedings*, 2022, pp. 791–800.

[20] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2002.

[21] A. Bangor, P. T. Kortum, J. T. Miller, Determining what individual SUS scores mean: Adding an adjective rating scale, Journal of Usability Studies 4 (2009) 114–123.

[22] B. Murat, Visualising Pyttern Pattern Matches, Master's thesis, Université catholique de Louvain, 2026.