

École polytechnique de Louvain

Visualising Pyttern Pattern Matches

Author: **Berkay MURAT**
Supervisor: **Kim MENS**
Readers: **Julien LIÉNARD, Benoît DUHOUX**
Academic year 2025–2026
Master [60] in Computer Science

Abstract

In programming courses, autograders are widely used to assess student code, especially when sizes are large. Providing targeted, high-quality feedback remains challenging in this setting. Detecting coding patterns and anti-patterns in submissions can support teachers in giving personalised feedback. Pyttern is a language that simplifies the definition and matching of such patterns in code written in Python or Java.

The existing Pyttern Visualizer is a web-based tool for debugging the Pyttern language by inspecting pattern matches; it is currently limited to one file and one pattern at a time. However, real-world teaching scenarios require multiple files and composition of multiple pattern files. Teachers who want to evaluate structural quality aspects of programs from many students at the same time lack structured support to run pattern detection across many submissions simultaneously and to inspect, organize, and filter the resulting matches.

This master thesis proposes an extended web-based visualization tool targeted at teachers that builds on the existing Pyttern Visualizer. The tool adds a section where users can compose patterns and analyze multiple code files at once, consult match results per file, and can open the Matcher view for any pattern–code pair. The solution is a refactoring and extension of the frontend web application of the Pyttern Visualizer that interacts with the existing Pyttern backend to run the matcher and present results in an overview, with filtering and navigation suited to teaching workflows. The extended Pyttern Visualizer supports teachers in giving pattern-based feedback at scale in programming education.

Acknowledgements

I would like to start by thanking Prof. Kim Mens and Julien Liénard for their support throughout this thesis. Our weekly meetings, their constant encouragement, and their thoughtful feedback were invaluable at every stage of this work. Their guidance helped me stay on track and pushed me to think more carefully about both the technical and academic dimensions of this project. I am also grateful for the opportunity they provided to conduct the usability test within the LINFO2335 course, which was essential for the evaluation of this work.

I also want to thank Dr. Benoît Duhoux for showing interest in my work and for accepting to read and be part of the jury for this thesis.

Additionally, I would like to thank the students who participated in the usability test for their time and their valuable feedback. Their input directly contributed to improving the tool.

Finally, I want to thank my family for their support throughout this journey. Both of my parents and especially my sister Rüya Murat have stood by me at every step and have been a constant source of encouragement throughout the thesis process and my academic journey.

Disclaimer

I would also like to acknowledge the use of generative AI tools in this work. ChatGPT was used exclusively for proofreading purposes, to improve clarity by rephrasing existing sentences. It was never used to generate original text or develop new ideas. Additionally, Claude models were used to generate synthetic test data as part of the evaluation described in Chapter 6, and as a coding assistant to help with refactoring boilerplate code. All implementation strategy, architectural decisions, and design choices are entirely my own.

1. Introduction.....	6
1.1 Context.....	6
1.2 Scope Clarification.....	7
1.3 Problem and Motivation.....	7
1.4 Objectives.....	9
1.5 Approach.....	10
1.6 Contributions.....	11
1.7 Roadmap.....	11
2. Background Materials.....	12
2.1 Pattern-Based Feedback in Programming Education.....	12
2.2 Human-Computer Interaction and User-Centered Design.....	14
2.3 Visual Design in Interactive Systems.....	16
2.4 Web Application Architecture.....	17
2.5 The React Technology Stack.....	18
3. Problem Statement.....	21
3.1 Limitations of the Existing Pyttern Visualizer.....	21
3.2 Problem Summary.....	23
4. Usage Scenario.....	25
4.1 Scenario Description.....	25
4.2 With the Current Pyttern Visualizer.....	26
4.3 With the Extended Pyttern Visualizer.....	27
5. Solution.....	29
5.1 Overview.....	29
5.2 Architecture.....	31
5.3 Compound Page Design.....	33
5.4 Filtering.....	40
5.5 Exporting Match Results.....	41
5.6 Matcher Integration.....	42
6. Software Validation and Maintainability.....	45
6.1 Software Maintainability.....	45
6.2 Software Testing.....	50

7. User Validation.....	56
7.1 Method.....	56
7.2 Results and Discussion.....	59
7.3 Threats to Validity.....	63
8. Conclusion.....	65
9. Future Work.....	66

CHAPTER 1

Introduction

In this chapter we outline the role of autograders and pattern-based feedback in introductory programming courses and clarify the scope of the Pyttern system and its existing visualizer, including its expected audience which is teachers. The restrictions of the current tool: single file and single pattern at a time, are described, together with a motivation for extending it to support compound patterns, multi-file matching, and export of match results in teaching scenarios.

1.1 Context

In introductory programming courses, autograders are commonly used to assess student code and scale well to large cohorts [1]. Correctness alone is often insufficient; students benefit from feedback on the structural aspects of their programs, such as which constructs and patterns or anti-patterns they used. Giving such feedback manually does not scale. Teachers need to recognise recurring patterns and anti-patterns across many submissions, which requires a way to express and detect these patterns in code and to see where they match.

Pyttern provides a language and matching engine for patterns in Python or Java files so that teachers can define patterns and match them against student code [2]. An existing web tool, the Pyttern Visualizer, was built for debugging the language rather than for teaching workflows. In this work, we want to extend the visualizer for teacher workflows without changing the Pyttern language or matcher.

Pyttern

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



Code

```
def foo():  
    if True:  
        x = 0  
    return x
```



Figure 1.1: Example of a simple Pyttern pattern with Python code.

1.2 Scope Clarification

The expected users are teachers in introductory programming, with experience in Pyttern, grading and experience in patterns and anti-patterns. This thesis does not modify the Pyttern language, compound pattern design, or the matching algorithm; those are taken as given. Pyttern is treated as an existing system with a stable interface for submitting patterns and code and obtaining match results. The developed tool is a client that uses this interface.

1.3 Problem and Motivation

Teachers who want to use Pyttern to support feedback in programming courses face a practical gap. The current Pyttern tooling does not provide a structured way to run pattern detection over many student submissions at once and then to inspect, organize, group, and filter the results in a clear and consistent manner. In practice, teaching involves many files and compound patterns, and the bottleneck for using Pyttern in such settings lies in how matches are presented and explored rather than in the language or matcher (Section 1.2).

The existing Pyttern Visualizer is geared toward a single pattern and a single code file at a time and does not support this multi-file, compound-pattern workflow.

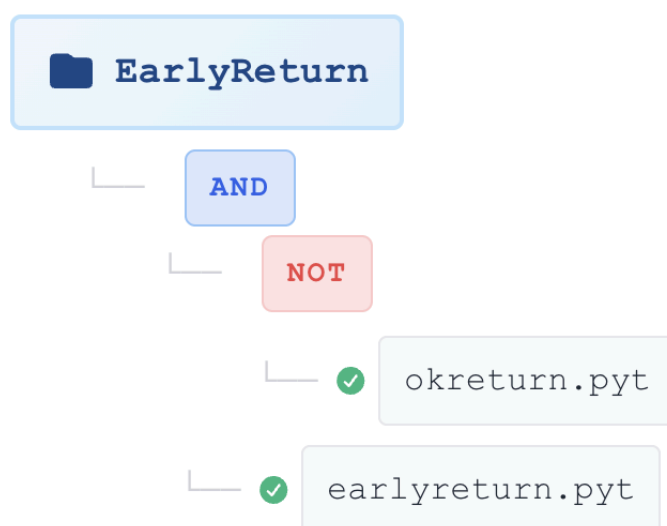


Figure 1.2: Example of a Compound Pattern

The current Pyttern Visualizer is insufficient for teaching because it was not designed around the tasks that teachers actually perform. When the goal is to run pattern detection over many student files and to work with compound patterns, that design forces a fragmented workflow. Teachers must run separate matches for each pattern and each file, record or remember results outside the tool, and re-upload and re-run whenever they want to compare or narrow down. The tool does offer a detailed view of a single pattern–code match, but there is no overview to navigate from and no way to filter the list of files by match outcome. From a user experience perspective, this makes it hard to understand results at a glance, to explore matches systematically, and to iterate (for example, to adjust a pattern and see how the set of matching files changes). The effort and cognitive load required to use Pyttern in real teaching settings therefore remain high [3, 4].

The gap is in the visualization layer, not in the underlying language or matcher. A tool that presents match results in an overview, allows filtering and sorting by pattern and outcome, and links that overview to the existing step-by-step debugging view would let teachers use Pyttern without manual bookkeeping and repeated re-runs. Improving how matches are presented and explored can therefore significantly increase the usefulness of

Pyttern in programming education and reduce the time and manual effort that teachers spend when giving pattern-based feedback.

1.4 Objectives

Given the problem and motivation described above, the aim of this thesis is to design and implement a teacher-oriented web application that extends the Pyttern Visualizer and makes it suitable for real teaching workflows. Teachers must be able to upload a compound pattern, and the uploaded pattern files should be validated before matching, and any validation errors must be shown clearly so that users can identify invalid patterns or code without leaving the tool. The application must validate the patterns and code files, and then run matching, present results in a structured way (per file and per pattern node), support efficient filtering and navigation, allow export of match results (e.g. to CSV) for record-keeping or further analysis, and offer a way to open any pattern–file pair in the existing Matcher view for step-by-step debugging.

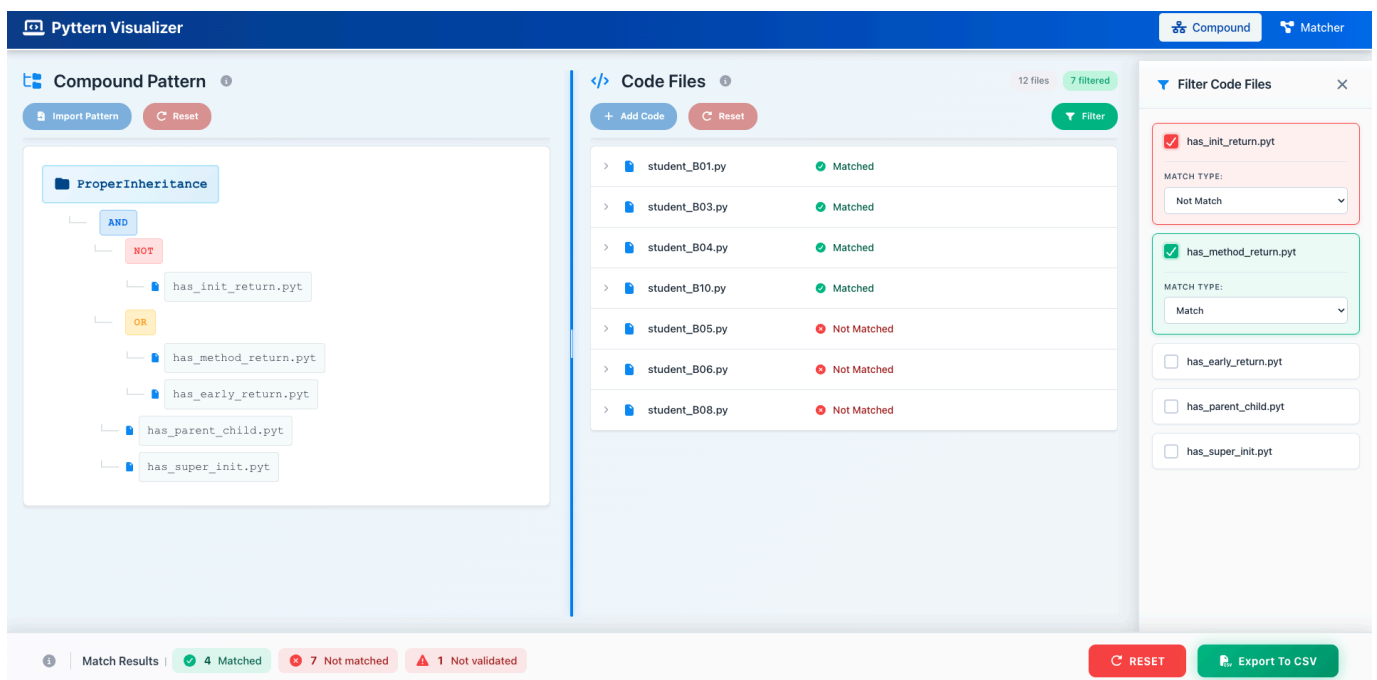


Figure 1.3 Screenshot of the extended Pyttern Visualizer (Compound view).

1.5 Approach

To design and implement the extended Pyttern Visualizer, initially, the necessary requirements are collected before proceeding to the design and architecture, and finally the implementation of the visualizer is done.

Requirements Elicitation

Initial requirements for the new interface were collected through discussions with the Pyttern Team. Functional user requirements were captured from early prototypes and supervisor feedback. Technical requirements were identified by analysing what the frontend needs from the backend (e.g. which endpoints and data structures are required to support the visualization). The matching logic and business rules of Pyttern were treated as a black-box; the goal was to identify which interface or endpoints to use rather than to modify them.

Design and Architecture

A new Compound page was designed by prototyping the frontend in Canva, based on a modified version of the current visualizer layout [5, 6]. These prototypes were used to gather feedback. Requirements were refined in light of the designed prototypes and the Pyttern Team input before implementation started.

The existing Pyttern Visualizer is implemented with React [7]. The rationale for keeping and extending React rather than switching to another framework such as Angular is discussed later in the thesis (Section 5.2) where the architecture is presented. Staying with React allowed building on the current codebase and reusing the existing Matcher view.

The React application was extended and refactored where necessary. The stack includes Bootstrap for layout and components, HTML and CSS for structure and styling, and Redux for application state management [8]. The Compound page feature was implemented on top of this base. The tool was evaluated to assess whether it supports teacher-oriented workflows and to inform any refinements. The evaluation method and findings are reported later in the thesis.

1.6 Contributions

This thesis makes the following contributions. The work delivers an extended version of the existing Pyttern Visualizer web app. The Pyttern backend and pattern language are left unchanged; all new behavior is implemented in the client, and the frontend architecture was refactored for extensibility (e.g. state management techniques, service layer, reusable API integration) so that further features can be added with less effort and without duplicating logic. The main contribution is implementing the new interactive visualization to address the need for compound pattern and multi-file matching. A usability evaluation was carried out and its findings are reported. The evaluation assesses support for teacher-oriented workflows, identifies strengths and weaknesses of the design, and informs directions for future improvement.

1.7 Roadmap

The remainder of the thesis is organized as follows. Chapter 2 presents the background material, covering pattern-based feedback in programming education, HCI and user-centered design, visual design considerations, web application architecture, and the React technology stack. Chapter 3 states the problem, describing the limitations of the existing Pyttern Visualizer and the requirements that the new tool must satisfy. Chapter 4 presents a usage scenario comparing the existing Pyttern Visualizer and the extended one. Chapter 5 describes the solution, including the overview, architecture, Compound page design, filtering, exporting match results as CSV, and Matcher integration. Chapter 6 discusses software validation and maintainability, covering the architectural and coding practices that support long-term maintainability and the automated test suite. Chapter 7 reports user validation, covering the method, the results and discussion, and the threats to validity. Chapter 8 draws the overall conclusions of the thesis. Chapter 9 outlines possible future work.

CHAPTER 2

Background Material

This chapter introduces the concepts and technologies that form the foundation of the work presented in this thesis. Section 2.1 covers the Pyttern language and the role of structural pattern detection in programming education. Section 2.2 presents the HCI and user-centered design principles that guided the design process of the Pyttern Visualizer interface. Section 2.3 discusses relevant visual design considerations for interactive systems, including the design of filtering interfaces. Section 2.4 explains the web application architecture concepts that apply to the system. Section 2.5 describes the React technology stack used in the implementation.

2.1 Pattern-Based Feedback in Programming Education

Feedback on code structure is harder to give at scale than correctness feedback, because it requires a way to express and detect patterns across many student submissions. This section introduces Pyttern as the language used for that purpose and explains why the right tools for presenting and exploring match results matter for teachers.

Pyttern: Language and Compound Patterns

Liénard, Mens, and Nijssen [2] present Pyttern, a program query language (PQL) designed to detect structural patterns and anti-patterns within source code. The name is a deliberate combination of "Python" and "pattern" and the language blends standard Python syntax with regular expression-like wildcards to reason about abstract syntax trees. Its primary goal is to balance expressiveness with usability so that non-expert users, such as teachers in introductory programming courses, can write structural patterns that are both easy to read and easy to write [2].

A central concept in Pyttern is the compound pattern, which allows teachers to group individual sub-patterns into hierarchical structures using logical operators such as AND, OR, and NOT. This makes it possible to check multiple coding criteria at the same time. For example, a teacher can define a single compound pattern that checks whether a student uses a specific loop construct while also avoiding certain global variables. While the Pyttern matching engine is capable of processing these compound structures, Liénard et al. [2] identify a need for improved debugging and visualization tools to help users work independently and reduce reliance on technical assistance. This thesis addresses that need at the interaction layer by refactoring and extending the visualizer to support compound workflows in a teacher-oriented interface.

Structural Feedback at Scale

As computing courses grow in size, autograding systems have become essential for handling programming assignments at scale [1]. However, checking only whether code runs correctly is often not enough for student development. Students also need feedback on how their code is structured, which helps them fix the root cause of their mistakes rather than just making failing tests pass [1].

Haldeman et al. [1] argue that giving meaningful structural feedback at scale requires more than pointing out local code issues. When feedback is too specific, students tend to patch the exact line that was flagged without understanding why it was wrong. More useful feedback connects observed patterns in student code to the broader concepts and skills they are expected to learn. For teachers, the main challenge is the lack of tooling support to run pattern detection across many submissions at once and to organize the results in a clear way. Without such support, it becomes very hard for teachers to spot common mistakes across all submissions and act on them in time.

Feedback Content Categories and Related Tools

Narciss [9] categorizes feedback content in learning environments beyond simple correctness signals. The most basic level, Knowledge of Result, only tells the learner whether a response was correct. Many traditional autograders stop here, limiting what a student can learn from a failing test. Narciss describes four more informative categories under the umbrella of elaborated feedback: Knowledge About Mistakes identifies the location or type of errors; Knowledge About How to Proceed offers a next step without revealing the solution; Knowledge About Task Constraints reminds the learner of

requirements such as using a specific construct; and Knowledge About Concepts explains the underlying subject matter.

A useful comparison is the Refactor Tutor by Keuning [10], an intelligent tutoring system that also applies patterns over abstract syntax trees to drive feedback. Students rewrite a functionally correct but low-quality program, and the system diagnoses each edit against a reference strategy. Feedback follows a layered hint tree that moves from a general question down to a concrete code change, giving students time to reason before revealing the answer.

The two tools solve different problems. The Refactor Tutor targets one student working on one file. The Pyttern Visualizer operates at both levels: its Compound view lets teachers analyze match outcomes across many submissions at once, while its Matcher view provides a detailed one-to-one inspection of a single file. In Narciss's terms, the Refactor Tutor delivers Knowledge About How to Proceed to students, while the Pyttern Visualizer surfaces Knowledge About Mistakes and Knowledge About Task Constraints to teachers, positioning this work as a teacher-facing analysis tool rather than a student-facing tutoring engine.

2.2 Human-Computer Interaction and User-Centered Design

Good interface design starts with understanding how users think and work. This section introduces the concepts that shaped the design of the extended Pyttern Visualizer, covering iterative and user-centered design methods and the usability evaluation approach used to assess the final tool.

HCI as a Science of Design

Carroll [3] defines Human-Computer Interaction (HCI) as a field that sits between psychology and social sciences on one side, and computer science and technology on the other. Over time, HCI has grown into a science of design that focuses on understanding how people interact with technology and making that interaction better. A key principle of modern HCI, as Carroll [3] describes, is moving away from fixed, step-by-step development toward iterative approaches, where designers build early versions, test them with real users, and update their goals based on what they learn. Within this framework,

usability engineering provides a practical way to measure and improve how well a system works for its users [3].

User-Centred Design and Design Thinking

Adhitya, Andreswari, and Alam [6] describe User-Centered Design (UCD) as a methodology that keeps the user's needs at the center of the design process to produce interfaces that are both useful and comfortable to use. The process typically follows four phases: understanding how users work, defining their requirements, designing the solution, and evaluating it against those requirements. Nasution and Nusa [5] present Design Thinking as a closely related approach, consisting of five iterative stages: Empathize, Define, Ideate, Prototype, and Test.

Prototyping is a key part of this process because it lets designers get user feedback early, before full implementation begins. Yatam [4] highlights horizontal prototyping as a useful technique, where the focus is on designing the visible layer of the interface, such as screens and menus, while the underlying logic is left out. This thesis follows this approach: prototypes were created in Canva and reviewed with the Pyttern team, which allowed the interface to be refined through several rounds of feedback before any coding started. As Nasution and Nusa [5] note, this cycle of prototyping, feedback, and refinement is what ensures the final interface actually fits the tasks users need to perform.

Usability Evaluation: The System Usability Scale

Nasution and Nusa [5] present the System Usability Scale (SUS) as a widely used method for measuring how satisfied users are with a software interface. It consists of a 10-statement questionnaire with both positive and negative statements. Each statement is rated on a scale from 1 (strongly disagree) to 5 (strongly agree) by the participants.

The final SUS score is calculated by combining the responses from all statements and scaling the result to a value between 0 and 100. A score of 90 or above is generally considered excellent, meaning users find the system easy and efficient to use [5]. Yatam [4] shows that comparing SUS scores from different versions of an interface is a practical way to check whether a redesign actually improved the user experience. The SUS methodology is used in Chapter 7 of this thesis to evaluate the extended Pyttern Visualizer.

2.3 Visual Design in Interactive Systems

Visual choices shape how easily users can read through information and notice what matters. This section covers two aspects of visual design that directly influenced the Pyttern Visualizer: how color can guide attention in an interface, and what principles should guide the design of a filtering feature.

Color and Visual Attention

Lewandowska and Olejnik-Krugly [11] study how color affects user behavior in graphical user interfaces, showing that it plays an important role in guiding where users look and what they notice. They describe two types of attention that color can support. Sustained attention refers to the state where a user focuses on one task for a long time, such as reading through a list of match results. Divided attention, by contrast, describes what happens when something in the interface needs to interrupt the user's current focus, such as an error message or a status change.

Their findings [11] show that certain colors work better for each type of attention regardless of the background. White is the most effective color for sustained attention because it is neutral and easy to read for long periods. For divided attention, green and yellow are the most reliable choices because they stand out clearly and users tend to notice them faster than other colors. Colors such as gray, orange, and violet depend too much on the surrounding context to be reliable and are therefore unsuitable for elements that carry important status information. These findings shaped the color choices in the Pyttern Visualizer, where a white background supports sustained reading and color indicators are used consistently to show match outcomes and validation states.

Designing Filtering Interfaces

The filter feature in the Pyttern Visualizer allows teachers to narrow down the visible file list to submissions that match specific combinations of sub-pattern outcomes. Teachers open a filter panel from the code files section header. The panel lists every leaf pattern from the compound tree, and for each one the teacher can choose whether to include it in the filter and whether the expected outcome should be a match or a non-match. When multiple conditions are set, only files that satisfy all of them stay visible. The section header always shows how many files are currently visible out of the total, so teachers can immediately see how much the filter has reduced the list. When no files match the active combination, an empty-state message is shown.

The filtering design in this thesis draws on the guidelines established by Bjäräng and Enderstein [12], who studied what matters when designing filtering for analytic software. The requirement that filter selections take effect immediately, and that the file list and visible-file count both update right away, follows their Guideline G6, which states that user interactions should produce instant feedback so the interface mirrors what the user is doing. The visible file count shown in the header is further supported by Guideline G1, where Bjäräng and Enderstein [12] suggest that showing the number of items matching a filter gives users a quick sense of the current state and helps them decide how to adjust the filter. The ability to combine several sub-pattern conditions and change them freely without resetting the session follows Guideline G5, which says users should be able to apply multiple filters at the same time and update them as they go. Finally, using plain labels such as "matched" and "not matched" in the filter panel follows Guideline G8, which recommends using plain language so users can express what they want without needing to know the underlying query syntax [12]. Together, these guidelines show that the filtering approach explained in Chapter 5 is consistent with established principles for building usable analytic interfaces.

2.4 Web Application Architecture

The architecture of a web application affects how responsive it feels and how easy it is to extend over time. This section covers the architectural concepts behind the Pyttern Visualizer, including the single-page application model that keeps the interface responsive and the layered structure that keeps the codebase organized.

Rich Internet Applications

Busch and Koch [13] describe Rich Internet Applications (RIAs) as a step forward in web development design to make web applications more interactive. Earlier web applications worked by sending every user action to the server and waiting for the server to reload the entire page in response. Rich Internet Applications change this by splitting the work between the server and the browser, so the interface can stay responsive while data is exchanged in the background [13].

This approach is the foundation of the Single-Page Application (SPA) model, where the entire application runs inside a single web page and feels much closer to a desktop application. Because the browser handles interaction and display logic directly, developers can build features such as live validation, drag-and-drop file upload, and real-time updates without reloading the page [13]. In the Pyttern Visualizer, this means teachers can upload

files, run matching, and explore results without the session being interrupted by page reloads.

Layered Architecture

Fowler [14] describes how separating an application into distinct layers keeps the codebase maintainable. A Presentation layer handles the interface, a Domain layer contains the business logic, and a Data Source layer manages communication with external services. The Service Layer pattern sits between these layers to coordinate responses to user actions, and Data Transfer Objects carry data between layers to keep communication clean [14]. The Pyttern Visualizer follows this principle through a six-layer structure described in Chapter 5.

2.5 The React Technology Stack

The Pyttern Visualizer is built on React, and the implementation decisions throughout this thesis follow from that choice. This section explains the core concepts of the stack, including the React component model, state management with Redux, maintainability practices, and the build and testing setup.

React: Component Model

Madsen, Lhoták, and Tip [15] describe React as a declarative, component-based JavaScript library for building user interfaces. An interface in React is built as a tree of components, where each component is a self-contained unit that holds its own input data (props), its own changing values (state), and a description of what it should display. Modern React development favors functional components and Hooks over older class-based components, as they allow stateful logic to be written more concisely and reused across different parts of the application [15, 7]. The Pyttern Visualizer extends an existing React codebase, and this component model made it straightforward to build new features on top of the existing ones without rewriting or duplicating the work already in place.

State Management with Redux

Pronina and Kyrychenko [16] compare state management approaches in React, describing Redux as a library for managing application state in one central place. The idea

is that all state lives in a single store, and the only way to change it is by sending an action, which is a plain object that describes what should change [16, 8]. The store processes these actions through reducers, which are functions that take the current state and an action and return the updated state. Because reducers always behave the same way for the same input, state changes are predictable and easy to test [16, 8].

React's built-in hooks are sufficient for managing state within a single component, but Pronina and Kyrychenko [16] find that Redux is a better fit when many parts of an application share the same state and need to stay in sync. That said, standard Redux requires a lot of repetitive code and can be slower than built-in hooks for small amounts of data. Bollu [17] notes that Redux Toolkit was introduced to address this, providing a simpler and more efficient way to write Redux logic. The Pyttern Visualizer uses Redux Toolkit because the compound workflow involves several pieces of state that depend on each other, including uploaded patterns, code files, validation outcomes, match results, and filter settings, all of which need to stay consistent as asynchronous operations complete. As described in the Redux documentation [8], Redux Toolkit introduces the concept of slices, where each slice groups a reducer and its related actions together in one place. The Pyttern Visualizer uses three slices to organize its state: one for the compound workflow, one for the matcher, and one for the global loading state.

Maintainability in React Applications

Bollu [17] defines maintainability in React as how easily a system can be changed to fix bugs, improve performance, or meet new requirements. Common sources of poor maintainability in React applications include components that have grown too large, components that are too tightly connected to each other, and state that is poorly organized. Bollu [17] discusses three metrics that are commonly used to track these issues. Lines of Code gives a rough measure of component size, though a large but well-organized component is not necessarily a problem. Cyclomatic complexity counts the number of decision branches in the code, and higher values tend to mean more bugs. Nesting depth measures how many levels deep a structure goes, where deeply nested code is harder to read and debug.

Bollu [17] recommends proactive approaches to keeping React applications maintainable. One of them is regularly refactoring components that have become too complex before the problems build up. These practices were applied during the refactoring work carried out in this thesis, as described in Chapter 6.

Build Tooling and Testing: Vite and Vitest

Vite is a modern build tool for web projects that focuses on making development fast [18]. It uses Hot Module Replacement (HMR) to update only the parts of the application that changed, without reloading the whole page, and it serves source code directly over native ES modules during development. For the Pyttern Visualizer, Vite also handles the connection between the frontend and the Flask backend by forwarding API requests under the `/api` prefix to the backend server, which keeps the development setup consistent with production.

To make sure the code works correctly, the project uses Vitest as its testing framework [19]. Vitest is built on top of Vite and shares its configuration, which means there is no need to set up a separate test environment. It works well with TypeScript and is used in the Pyttern Visualizer to run unit tests on utility functions, Redux slices, and service logic, as well as integration tests that check the full matching pipeline against the backend API.

CHAPTER 3

Problem Statement

This chapter states the problem that the work in this thesis addresses. The problem exists in the context of the Pyttern and its existing visualization tool. Section 3.1 describes the limitations of the current Pyttern Visualizer from the perspective of teachers in programming courses. Section 3.2 summarizes the problem statement that guides the design and implementation presented in later chapters.

3.1 Limitations of the Existing Pyttern Visualizer

The existing Pyttern Visualizer was built as a debugging tool for the Pyttern language. Its web interface is organized around a single pattern and a single code file. A user uploads one pattern file on the left side and one code file on the right side, then steps through the matching process to observe how the pushdown automaton processes the pattern against the code. This design serves its original purpose well, since Pyttern developers need to inspect individual matches in detail to verify and debug the matching algorithm.

The limitation lies in this web interface rather than in Pyttern itself. The Pyttern language and its matcher already support compound patterns, which group several sub-patterns into a single hierarchical structure using logical operators such as AND, OR, and NOT [2], and they can apply such patterns to multiple files through the command line interface. The way compound patterns are structured and displayed as a logical tree is described in detail in Section 5.3. The current Pyttern Visualizer, however, exposes only the single-pattern-single-file workflow, because it was built solely for debugging Pyttern. A teacher working through the visualizer cannot upload a folder-based compound pattern and run it as a unit, and cannot analyse a batch of student submissions in one session, even though the underlying matcher is able to do both.

In a typical teaching scenario, a teacher works with a compound pattern and needs to apply it to many student submissions at once. For example, a teacher may want to check that a student uses a specific loop construct, avoids certain global variables, and calls a

required function [1]. Rather than checking each criterion separately, the teacher can define one compound pattern that captures all the requirements together and apply it to every submission. This kind of batch assessment is the primary usage scenario that motivates this work.

Because the website does not expose these capabilities, a teacher who relies on it is pushed into a manual workflow. They must upload each pattern file separately, match it against each student file one at a time, record results outside the tool, and then combine these individual outcomes by hand to decide whether the compound pattern as a whole matched. This has to be repeated for every submission. The website offers no overview of how a set of files performed against a compound pattern, no way to filter submissions by match outcome, and no way to export results for record keeping.

To support batch assessment through the visual tool, several tasks would need to be possible that the current website does not offer. A teacher needs to import a compound pattern as a structured folder and see it displayed as a logical tree so that its structure is immediately visible. They need to upload multiple student code files, run the compound match in one action, and see the results organized per file and per sub-pattern. They need to filter the result list to find, for example, all submissions where one pattern matched while another did not. They need to export the match results for use outside the tool. And they need a way to move from the batch overview to a detailed step-by-step inspection of a specific pattern against a specific file, without re-uploading anything.

Figure 3.1 shows the interface of the existing Pyttern Visualizer. The layout is designed for one-to-one inspection, with a single pattern editor on the left, a single code editor on the right, and step navigation controls at the bottom. The bottleneck therefore lies in the visualization and interaction layer of the website rather than in the Pyttern language or matcher. Improving how compound matches are presented, navigated, filtered, and exported would make Pyttern significantly more useful for teachers without requiring any changes to the Pyttern language or its matching algorithm.

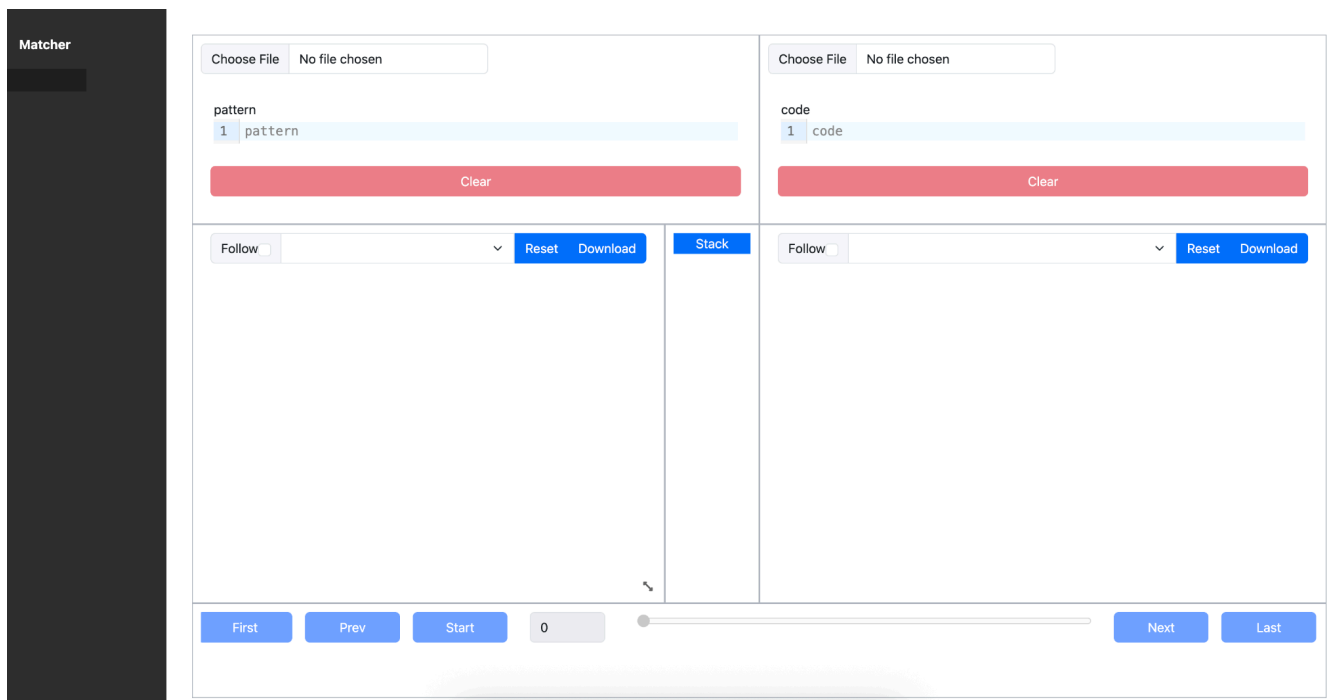


Figure 3.1 The Existing Pyttern Visualizer (Matcher View)

3.2 Problem Summary

The requirements for the new tool were gathered through meetings with the Pyttern team. In these meetings, the team first explained how Pyttern works at a high level, then identified the problem explained in the previous sub-sections. Based on this input, the initial requirements were specified and used to prepare a set of mockup UI designs. These designs were then reviewed together with the Pyttern team, refined through discussion, and finalized into the following requirements.

R1. Compound Pattern Import and Visualization. The tool must allow teachers to import a compound pattern as a structured folder and display its hierarchy as a logical tree showing the sub-patterns and their operators (AND, OR, NOT). This gives teachers an immediate overview of the pattern structure before any matching takes place.

R2. Multiple Code File Upload. The tool must allow teachers to upload multiple student code files in a single session. This removes the one-file-at-a-time constraint of the existing tool and makes assessment at the scale of a typical course feasible.

R3. Input Validation. The tool must validate uploaded pattern folders and code files before running a match. This means detecting missing files, unsupported formats, or structural inconsistencies and reporting them clearly, so that errors are caught before a match run begins rather than during it.

R4. Batch Compound Matching. The tool must run the compound match across all uploaded code files in one action, applying the full compound pattern to each file using the existing Pyttern backend without modifying the matching algorithm or language.

R5. Structured Result Overview. The tool must present match results organized per code file and per sub-pattern, giving teachers a clear view of how each student submission performed against both the compound pattern and its individual components.

R6. Result Filtering. The tool must allow teachers to filter the result list by match outcome per sub-pattern. This makes it possible to quickly identify, for example, all submissions where a specific criterion matched or did not match.

R7. Export Match Results. The tool must support exporting match results in a format suitable for use outside the tool, such as for grading or record-keeping.

R8. Integration with the Existing Matcher View. The tool must preserve the existing single-pattern Matcher view and allow teachers to navigate from the batch result overview to a detailed step-by-step inspection of a specific sub-pattern match against a specific file, without having to re-upload any files.

CHAPTER 4

Usage Scenario

This chapter presents a concrete scenario that illustrates the practical difference between the current Pyttern Visualizer and the extended tool developed in this thesis. The scenario is drawn from a realistic teaching context and demonstrates both the limitations of the existing tool and the improvements offered by the new Compound page.

4.1 Scenario Description

Consider an advanced Python programming course in which a teacher has assigned students to implement a Vehicle hierarchy. Each student is expected to define a base *Vehicle* class and at least one subclass (for example, *Car*, *Truck*, or *Bus*) that inherits from it. To receive a good score, a submission must satisfy a set of structural requirements that the teacher has formalized as pattern files using Pyttern [2]. Specifically, a valid submission must contain a parent class and a child class that inherits from it, the child class must call *super().__init__()* in its constructor, the *__init__* method must not return a value, and the code must include at least one method that either returns a value directly or contains an early return inside a conditional block.

To address these requirements, five Pyttern files were prepared. Table 4.1 provides an overview of each file and the structural property it is designed to detect.

Pyttern File	What it detects
has_parent_child.pyt	A parent class and a child class that inherits from it
has_super_init.pyt	A child class that calls <code>super().__init__()</code> in its constructor
has_init_return.pyt	A class whose <code>__init__</code> method returns a value
has_early_return.pyt	A class with a method that has a return inside a conditional block
has_method_return.pyt	A class with a method that has a direct return statement

Table 4.1 Pyttern Files and Their Detection Targets

Together, these patterns define a compound rule called *ProperInheritance*, structured as follows. The code must match: *has_parent_child* AND *has_super_init* AND NOT *has_init_return* AND (either *has_early_return* OR *has_method_return*). The teacher has received 12 student submissions and wants to check which of them satisfy this compound rule. Evaluating this many submissions against this many criteria manually does not scale [1].

4.2 With the Current Pyttern Visualizer

The current Pyttern Visualizer operates on a single pattern file and a single code file at a time. There is no notion of compound patterns in the interface; each pattern must be submitted and evaluated independently. To work through this scenario using the current tool, the teacher would need to upload each of the five pattern files one by one and check each of the 12 student submissions against it separately. That amounts to 60 individual match operations, each requiring a separate upload and a separate inspection of the result. The results would need to be recorded externally, for example in a spreadsheet, so that the teacher can later combine them manually and apply the logical operators (AND, NOT, OR) by hand to determine whether each student satisfies the full compound rule.

Beyond the number of operations, the current tool provides no overview. There is no way to filter or compare submissions side by side, cross-referencing results requires external bookkeeping, and adjusting a pattern means re-running all relevant matches from scratch. As a result, the teacher ends up spending more time managing the tool than analyzing the results.

4.3 With the Extended Pyttern Visualizer

The extended tool introduces a dedicated Compound page that is designed around exactly this kind of scenario. The teacher begins by organizing the five pattern files into the compound folder structure that encodes the logical rule, mirroring the ProperInheritance tree with AND, NOT, and OR operator folders containing the appropriate .pyt files. This folder is then uploaded once to the Compound page as shown in Figure 7.1. All 12 student code files are uploaded together in a single step. After clicking Match, the tool sends the necessary requests to the backend and displays the results in a structured overview.

Each student submission appears as a card showing its overall compound match result (matched, not matched, or not validated) along with the individual result for each of the five pattern nodes (Figure 5.8). The teacher can immediately see that 4 submissions matched the full compound pattern, 7 did not, and 1 could not be validated because of syntax errors in the submitted file. The not-validated submission is clearly distinguished from the others, and the tool displays the validation error so the teacher can understand what went wrong without leaving the page.

The filter panel allows the teacher to narrow the view by compound result or by the outcome of any individual pattern node. For example, the teacher can filter for submissions that matched *has_parent_child* but did not match *has_super_init* to identify students who implemented inheritance but forgot to call the parent constructor. This kind of targeted inspection, designed to give immediate feedback as the teacher adjusts filter conditions, follows established principles for analytic filtering interfaces [12]. If the teacher wants to look at the step-by-step match details for a particular pattern–file pair, clicking the corresponding entry opens that pair directly in the Matcher view, using the same familiar interface that already existed in the current tool. No re-uploading is needed.

The contrast between the two workflows is significant. What required 60 separate operations, manual record-keeping and logical reasoning in the current tool is reduced to a single upload, an automatic match and exportable results, and a navigable overview in the

extended tool. The teacher can focus on interpreting results and giving feedback rather than managing the mechanics of running the tool.

This scenario, with its 12 student submissions and five-pattern compound rule, formed the basis of a task in the usability test conducted to evaluate the extended Pyttern Visualizer. The usability test, its methodology, and its findings are explained in Chapter 7.

CHAPTER 5

Solution

This chapter presents the design and implementation of the Pyttern Visualizer developed in this thesis. The tool is a web-based frontend for the existing Pyttern pattern matching language. It supports two main activities. First, users can apply compound matches matching over several source code files and inspect outcomes per file and per pattern. Second, which already exists, users can step through how the underlying pushdown automaton matcher processes a single pattern against a single code file.

This chapter is organized as follows. Section 5.1 gives a high-level overview of the two main views and the end-to-end workflow on the Compound page. Section 5.2 describes the client-server architecture, the division of work between frontend and backend, and some implementation details on how the application is built such as state management with Redux. Section 5.3 details the Compound page design, including the compound pattern tree, the code file list, the match toolbar, and the information and confirmation dialogs. Section 5.4 covers filtering of match results. Section 5.5 explains how match results can be exported as CSV files. Section 5.6 describes how users open a pattern–file pair in the Matcher view.

5.1 Overview

The Pyttern Visualizer adds an interactive graphical layer on top of Pyttern so that teachers can work in the browser instead of relying only on command-line use. The application is organized around two views, both reachable from the top navigation bar.

The new Compound page is the main page of the application. It supports batch analysis of several source code files against a hierarchical compound pattern. Users import a folder that represents the compound pattern tree, including logical operators AND, OR, and NOT and leaf pattern files in Pyttern syntax [2]. They add one or more source code files, validate patterns and code, run the match, and inspect per-file results. Results can be

filtered by pattern and match type, exported as CSV, or opened in the Matcher view for step-by-step inspection of a single pattern against a single file.

There is also the Matcher page which used to be the main page of the existing Pyttern Visualizer. It offers a detailed, step-by-step view of how the pushdown automaton-based matcher processes one pattern against one code file by using the pyttern [2]. Users provide a pattern and a code file, then move through match steps while observing abstract syntax trees, automaton state, stack changes, and highlighting of the current position in the code. Since this view is much more technical and detailed than the Compound page, the latter was promoted as the main page of the Visualizer.

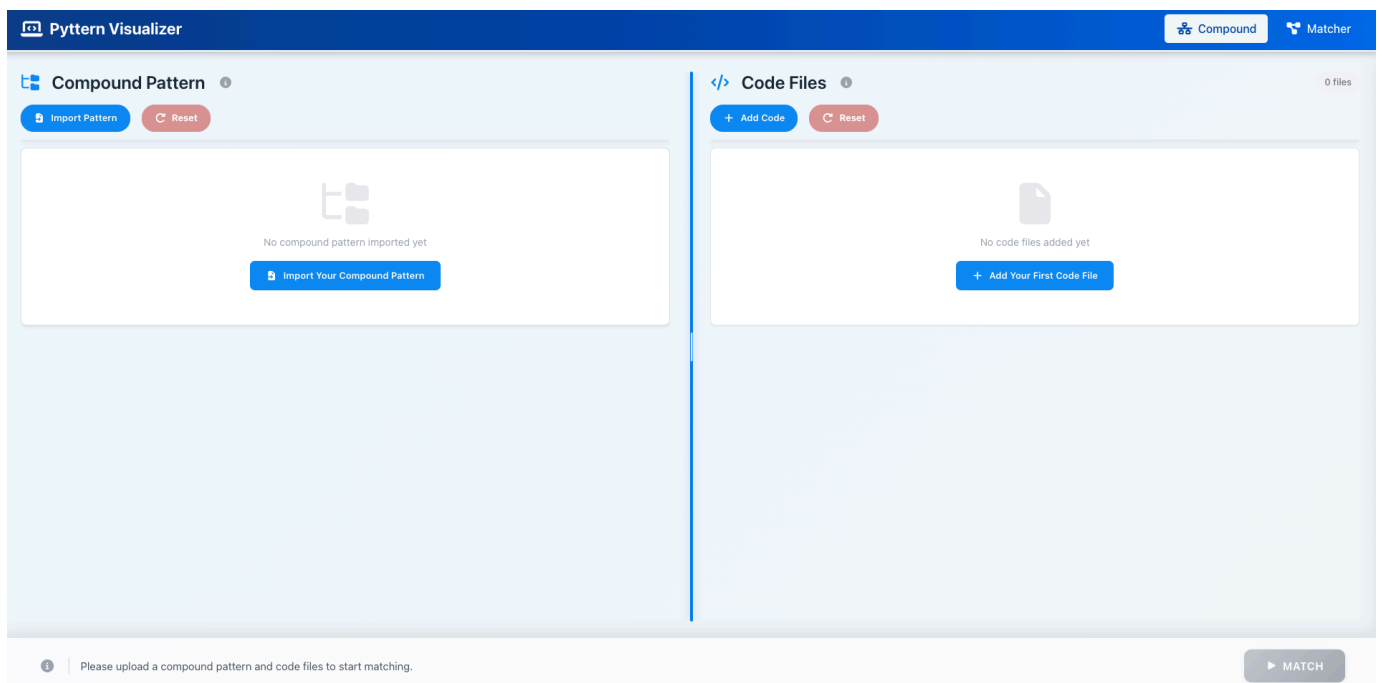


Figure 5.1 Initial State of the Compound Page

The typical workflow on the Compound page proceeds in several stages. Figure 5.1 illustrates the initial state of the Compound page before any import operation, while Figure 5.4 shows the same page after a compound pattern folder and code files have been imported. The user opens the Compound page and imports a compound pattern folder from the local file system using the browser's file system access API. The application parses the folder into a tree and distinguishes operator directories from leaf pattern files. The user also adds source code files to the code files section. When there is a new uploaded code file or compound pattern, the application sends the content of patterns and

code files to the backend for syntax validation. When inputs are ready, the user starts matching by clicking the Match button on the bottom right. The backend runs compound matching and returns per-file and per-pattern results. Each code file appears as a card with an overall match status (Figure 5.8). Expanding a card shows the pattern tree for that file with per-pattern match indicators. The user may filter the file list, export results, or click a pattern leaf to open the Matcher view with that pattern and file preloaded.

This workflow connects Pyttern’s analytical capabilities to workflows that teachers need in practice, where multiple submissions and patterns need to be reviewed without repeatedly switching tools or re-uploading the same content by hand. A concrete demonstration of this workflow in a realistic teaching scenario is presented in Chapter 4.

5.2 Architecture

The Pyttern Visualizer follows a client-server architecture, separating the pattern logic from the user interface. The backend remains the existing Python Flask REST API provided by the Pyttern project and exposes endpoints for validation and matching. Since this API was already in place, the frontend was designed to integrate with it rather than introducing a separate server layer.

For the frontend, a single-page application approach was chosen to enable a responsive, interactive experience without full page reloads [13] which is important for a tool where users iterate quickly on pattern input and immediately inspect results.

For the framework choice, an early option was to build a new frontend with Angular, because the author had more prior experience with that framework. However, the existing Pyttern Visualizer already implemented the Matcher page in React, including editors, graphs, step control, and integration with the backend. Reimplementing those features in Angular would have duplicated a large amount of work and created a risk of differences in behavior. The decision was therefore taken to adopt React [15] for this project, extending the existing codebase. Since the existing implementation already used TypeScript alongside React, and the author had prior experience with TypeScript through Angular development, TypeScript was adopted for the new frontend code as well, keeping the codebase consistent and taking advantage of the strong TypeScript support in the React ecosystem.

Additionally, Vite [18] was chosen as the build tool because of its faster development server and hot module replacement. During development, Vite is configured to proxy

requests under the /api prefix to the Flask server, eliminating the need for CORS (Cross-Origin Resource Sharing, a browser security mechanism that blocks requests made to a different origin than the serving domain) configuration and keeping the development environment consistent with production.

The Pyttern Visualizer follows a client-server architecture in which pattern matching logic runs on the server and the user interface runs in the browser as a single-page application. Figure 5.2 illustrates this structure.

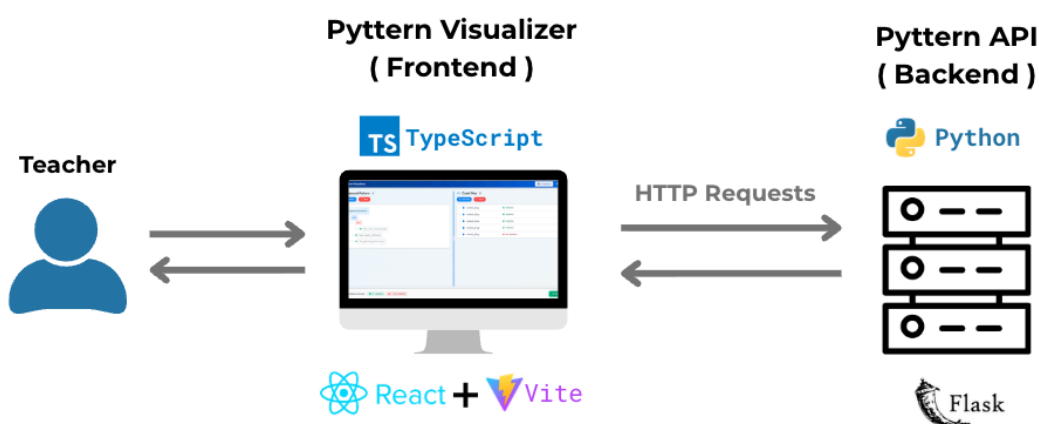


Figure 5.2 High-Level System Architecture

The backend is the existing Python Flask REST API provided by the Pyttern project. It exposes endpoints for syntax validation and pattern matching that were already in use for the Matcher page. For the Compound page, two additional endpoints were introduced in collaboration with the Pyttern team. The first endpoint accepts a set of pattern and code files and returns a validation result for each one individually. The second takes the full compound pattern together with the uploaded code files and returns a match outcome per file and per sub-pattern. This per-sub-pattern data is what drives the annotated pattern trees shown in the file cards and the filter feature described in Section 5.4.

State management uses Redux [8, 16] to keep compound and matcher state consistent across the application. Several aspects of the compound workflow depend on each other, including imported patterns, uploaded code files, validation outcomes, match results, and filter settings. Centralising this state in a shared store ensures that asynchronous operations such as validation and matching leave the interface in a consistent state. Redux thunks handle the sequences of API calls and state updates so that loading indicators and action

availability remain aligned with the current workflow stage. Figure 5.3 illustrates the general Redux data flow.

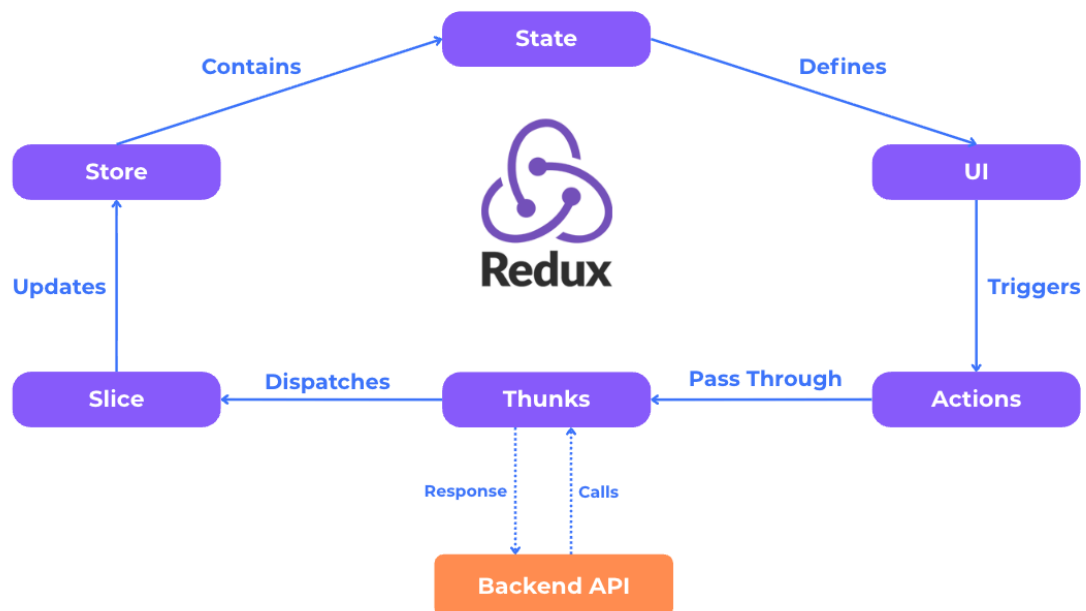


Figure 5.3 Redux Data Flow Architecture

5.3 Compound Page Design

The Compound page is the central feature of this thesis. It is built as a split layout with a persistent footer toolbar. Figure 5.4 shows the page after a compound pattern folder and several code files have been imported.

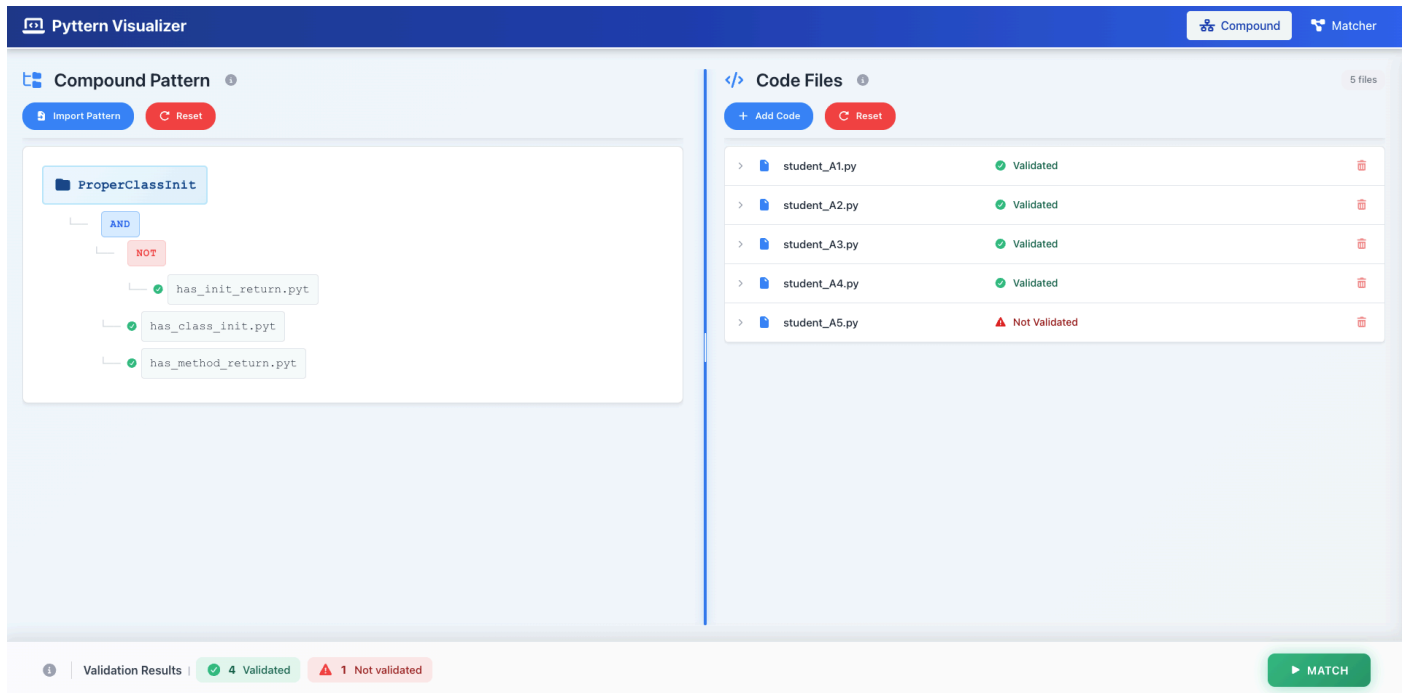


Figure 5.4 Compound Page After Import Operations

Compound Pattern Section

The compound pattern section displays the imported folder as a recursive tree. Operator directories (AND, OR, NOT) appear as internal nodes with distinct colors so that the logical structure of the pattern is immediately visible [11]. Leaf pattern files appear at the leaves of the tree. Before matching, each leaf shows an icon that reflects its current validation state. After matching, the icons are updated to indicate whether that sub-pattern matched. Figure 5.5 shows the pattern section after import where one leaf pattern has failed validation, meaning the backend detected a syntax error in the file that prevents it from being used for matching.

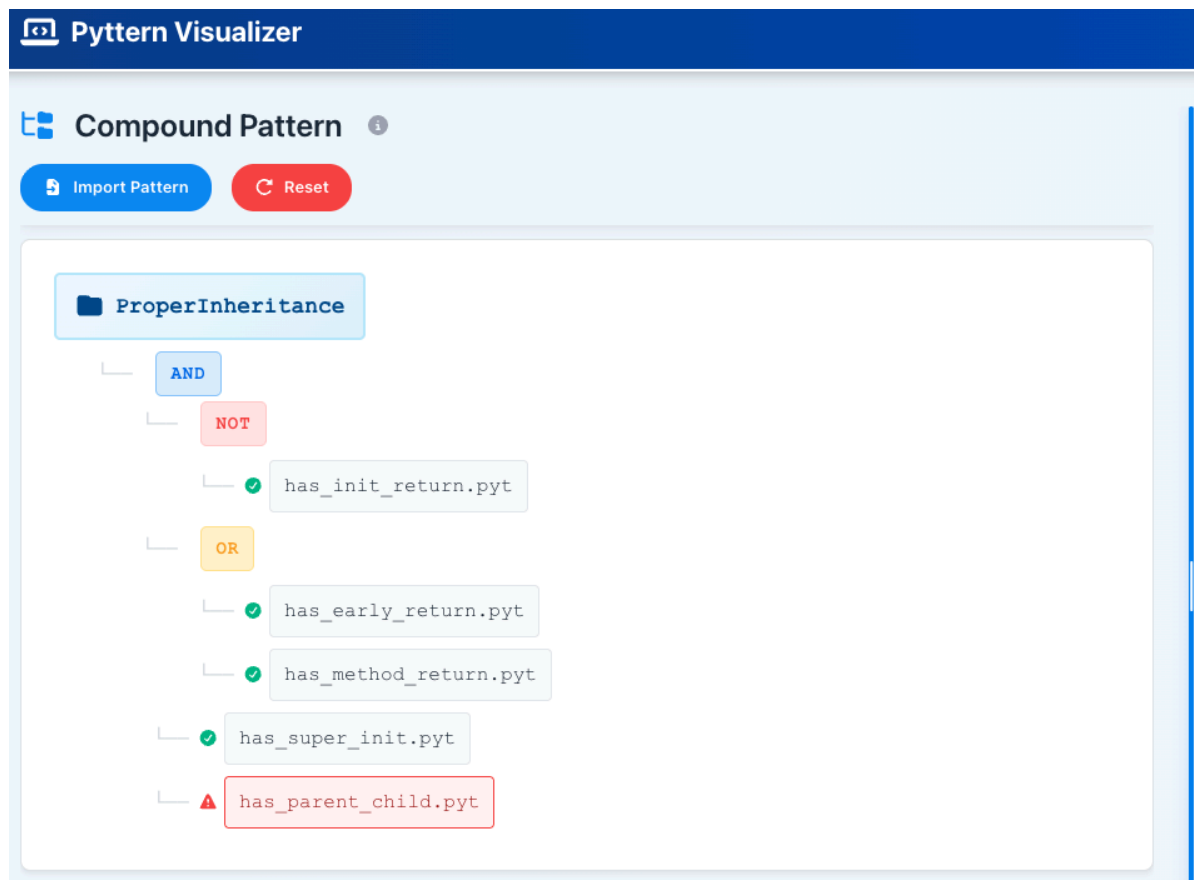


Figure 5.5 Compound Pattern Section with an Unvalidated Leaf Pattern

Teachers can click any leaf to open a detail modal that shows the pattern source with syntax highlighting and any validation errors, without navigating away from the page. Figure 5.6 shows such a modal for the leaf pattern of Figure 5.5 that has not yet been validated.

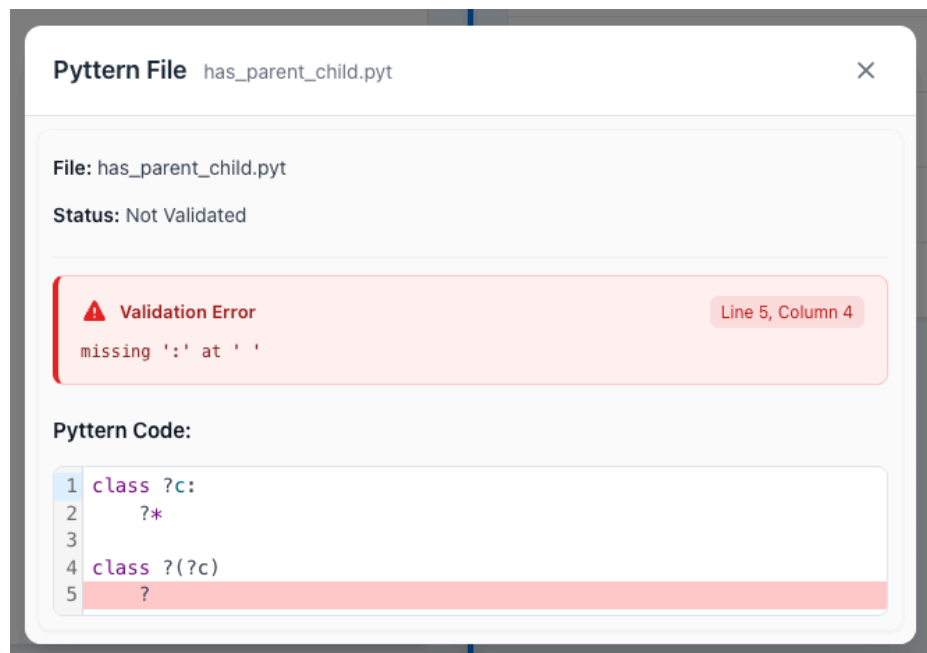


Figure 5.6 Detail Modal for a Pattern File

Code Files Section

The code files section presents each submission as an expandable card. The card header shows the filename and a status badge. Before matching, badges distinguish validated files, files with syntax errors, and files that are still pending validation. After matching, the badges reflect the overall compound match outcome for each submission. Figure 5.7 shows the code files section after a completed match.

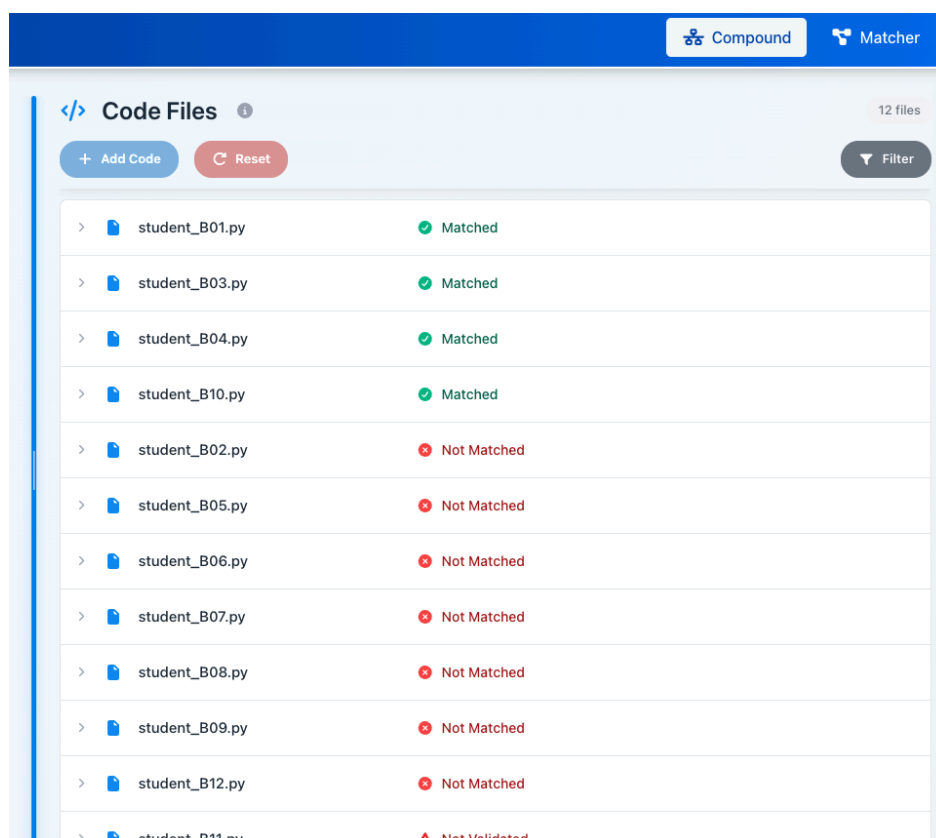


Figure 5.7 Code Files Section After Matching

Expanding a card reveals the compound pattern tree annotated with per-leaf outcomes for that specific file. This view uses colors and icons consistently to distinguish matched and not-matched sub-patterns, giving teachers a clear picture of why a submission received its overall result. Figure 5.8 shows an expanded card with per-pattern results.

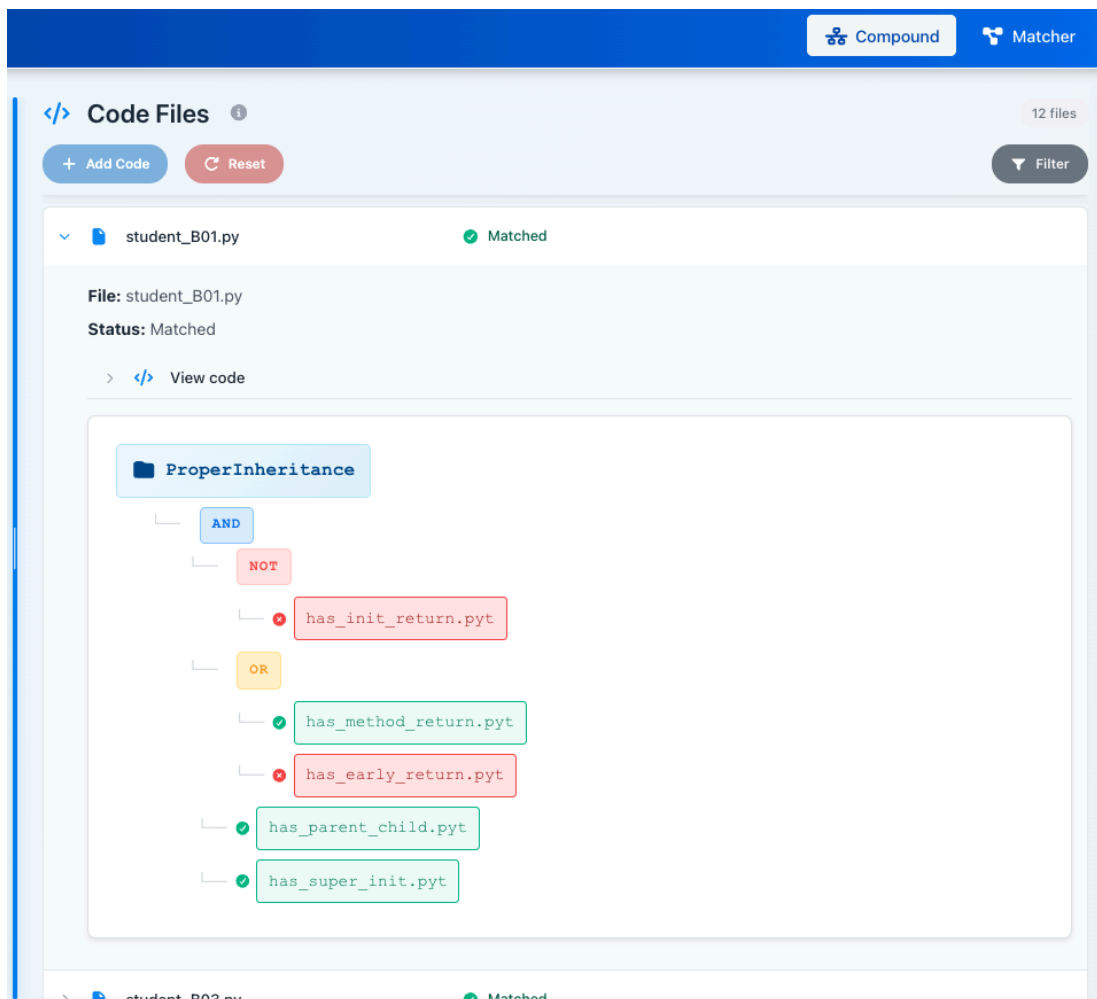


Figure 5.8 Expanded Code File Card with Per-Pattern Match Results

Match Toolbar

The footer toolbar coordinates the overall matching workflow. Before any files are loaded, it prompts the teacher to import a pattern and add code files. As files are added, the toolbar reports validation counts of the uploaded code files, as shown at the bottom of Figure 5.4. It also highlights any prerequisites that are not yet satisfied, such as pattern files that fail validation. The Match button becomes active only when both the compound pattern is uploaded with all the pattern files validated and at least one code file is uploaded.

After matching, the toolbar summarizes outcomes across all uploaded files, indicating how many matched, did not match, could not be validated, or produced an unexpected error. It also provides the Reset action, which clears the current session so a new batch can

begin, and the Export to CSV action described in Section 5.5. Figure 5.9 shows the toolbar in its post-match state.

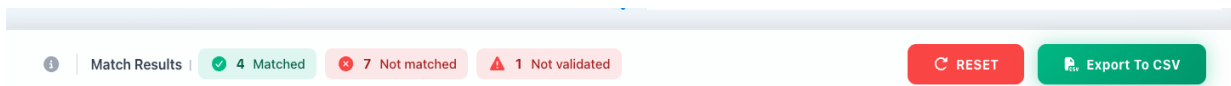


Figure 5.9 Match Toolbar After Matching Process

Information Modals and Confirmation Dialogs

Each main section of the Compound page includes an information button that opens a short modal dialog explaining the purpose of the section and the expected interaction flow. Figure 5.10 shows the modal for the compound pattern section. These modals reduce the need for external documentation and help first-time users understand the expected workflow without leaving the page.

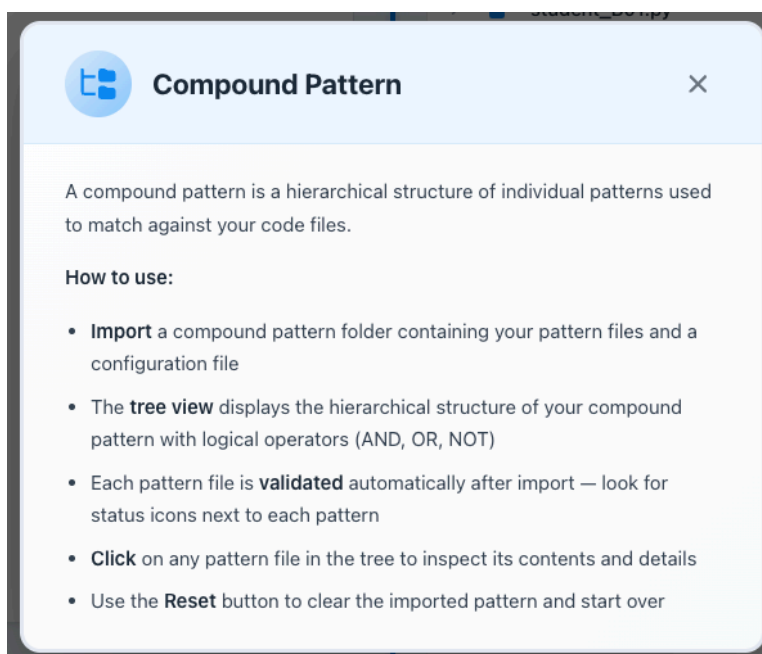


Figure 5.10 Information Modal of the Compound Pattern Section

Destructive actions such as resetting the pattern, resetting the code file list, or clearing the session after matching are guarded by confirmation dialogs that ask the teacher to

confirm before any data is discarded. The application also detects an unsaved state when the teacher navigates away and shows a prompt to prevent accidental loss of a matching session in progress. Figure 5.11 shows the confirmation modal for trying to leave the page after inserting a file.

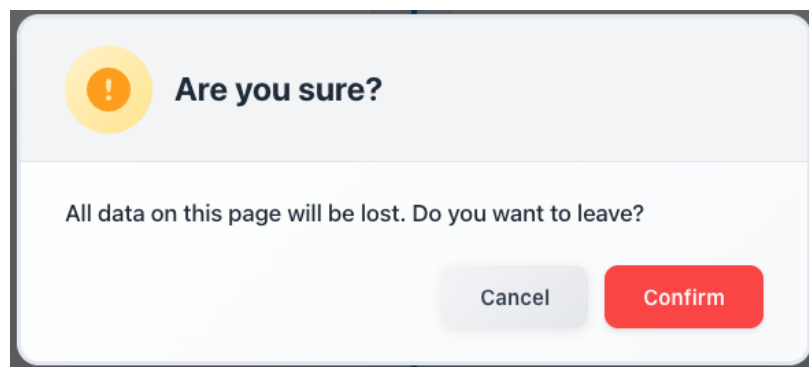


Figure 5.11 Confirmation Dialog for Trying to Navigate to Another Page

5.4 Filtering

When a large number of code files have been matched, reviewing every result individually is time-consuming. The filter feature allows teachers to restrict the visible file list to submissions that satisfy specific combinations of sub-pattern outcomes [12]. This makes it practical to locate, for example, all files where a particular pattern matched while another did not.

Filtering runs entirely in the browser over data that has already been returned by the match. Teachers open a filter panel from the code files section header. The panel lists every leaf pattern from the compound tree. For each pattern, the teacher can choose whether to include it in the active filter and whether the required outcome should be a match or a non-match. Defaults follow the NOT context on each leaf so that patterns under NOT start from the expectation of non-match unless the teacher changes them. When multiple patterns are selected, only files that satisfy all active conditions remain visible. The section header updates to show how many files are currently visible out of the total, and an empty-state message appears when no files satisfy the current combination of constraints. Figure 5.12 shows the filter panel alongside the resulting file list after a filter has been applied.

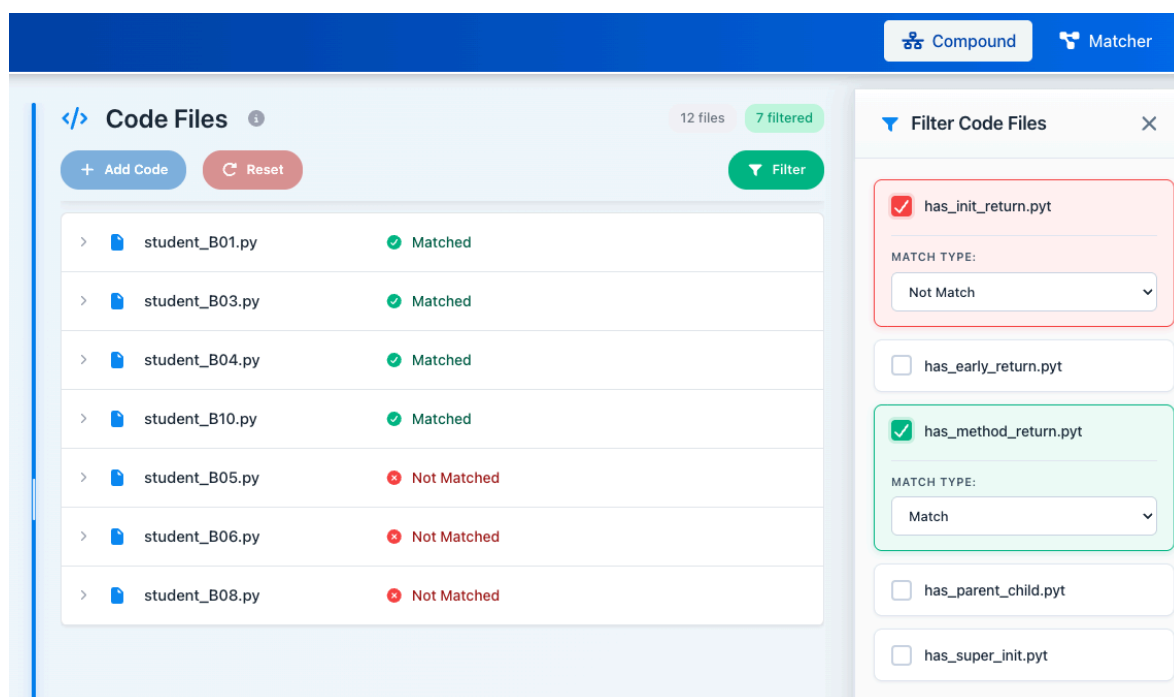


Figure 5.12 Filter Panel and Code Files After Applying a Filter

5.5 Exporting Match Results

Teachers often need to record or share batch results outside the browser, for example to compile grades, pass results to a grading script, or include them in a report. After a compound match, the match toolbar provides an option to export results as a CSV file. The file is generated in the browser from the same data structures that drive the interface, so the exported content is always consistent with what the teacher sees on screen.

Each row in the exported file corresponds to one code file. The first column holds the filename, the second states the overall match outcome for that file, and the remaining columns record the outcome for each leaf pattern in the compound tree. Pattern columns are ordered alphabetically by filename so that repeated exports from the same compound pattern produce a consistent header row. Figure 5.13 shows an example of an exported file.

Code File Name	ProperInheritance	has_early_return.pyt	has_init_return.pyt	has_method_return.pyt	has_parent_child.pyt	has_super_init.pyt
student_B01.py	matched	not-matched	not-matched	matched	matched	matched
student_B03.py	matched	not-matched	not-matched	matched	matched	matched
student_B04.py	matched	not-matched	not-matched	matched	matched	matched
student_B10.py	matched	not-matched	not-matched	matched	matched	matched
student_B02.py	not-matched	not-matched	not-matched	not-matched	matched	matched
student_B05.py	not-matched	not-matched	not-matched	matched	matched	not-matched
student_B06.py	not-matched	matched	not-matched	matched	not-matched	not-matched
student_B07.py	not-matched	not-matched	not-matched	not-matched	matched	matched
student_B08.py	not-matched	not-matched	not-matched	matched	matched	not-matched
student_B09.py	not-matched	not-matched	not-matched	not-matched	not-matched	not-matched
student_B12.py	not-matched	not-matched	not-matched	not-matched	matched	matched
student_B11.py	not-validated	not-validated	not-validated	not-validated	not-validated	not-validated

Figure 5.13 Example of an Exported Match Results CSV

5.6 Matcher Integration

The Compound page answers batch questions about which submissions matched and how the sub-patterns contributed. The Matcher page answers detailed questions about a specific pattern-file pair, showing the step-by-step execution of the underlying pushdown automaton. The two views are connected so that teachers can move from a batch result to a detailed inspection without any manual copying or file management.

After matching, each leaf in an expanded file card is clickable. Clicking a leaf opens the Matcher page in a new browser tab with the selected pattern and code file already loaded. The teacher can keep the batch overview open in the original tab while investigating an individual result in the new tab. The Matcher page arranges its tools in a resizable layout. The top area holds two editors side by side for the pattern and the code file. The central panels show the pattern abstract syntax tree, the automaton stack, and the code tree or automaton graph. Step navigation at the bottom allows the teacher to move through the match one step at a time. At each step, the relevant node in the graph is highlighted, the stack reflects the current automaton state, and the corresponding position in the code editor is marked. Figure 5.14 and 5.15 show the Matcher page before and during step navigation.

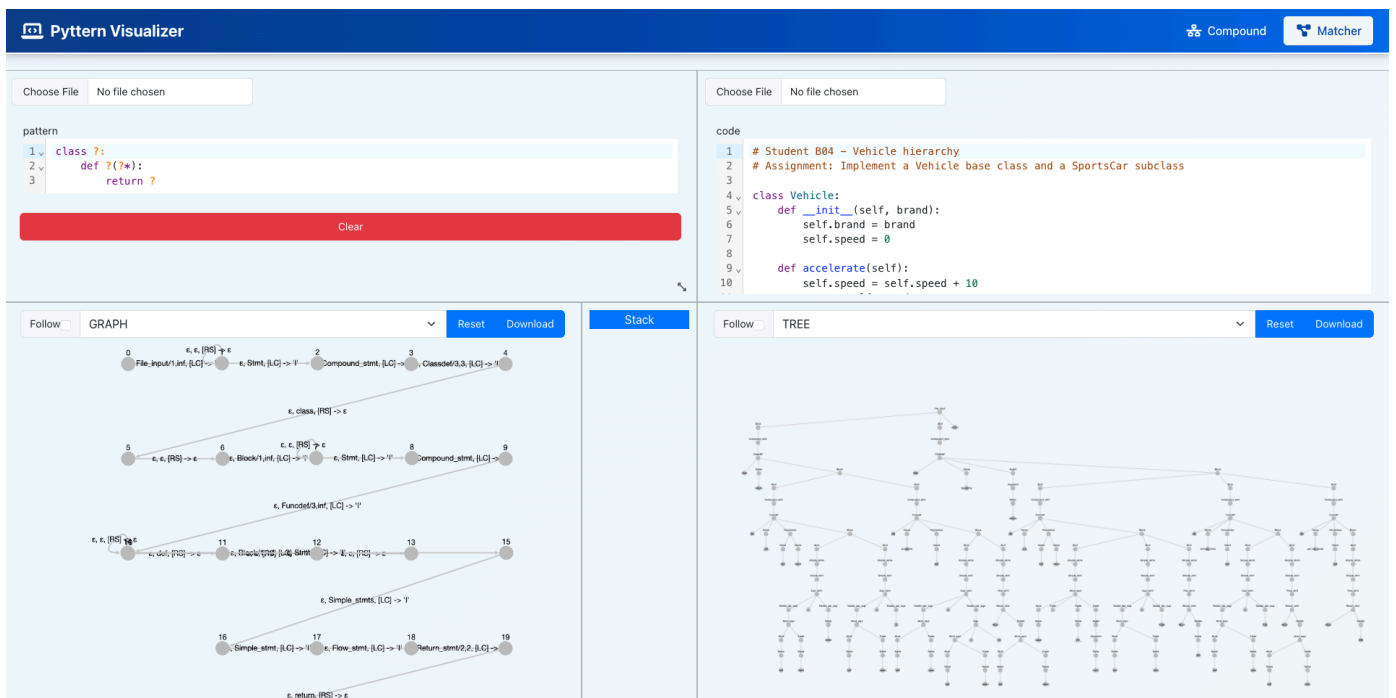


Figure 5.14 Matcher Page With Pattern and Code Files

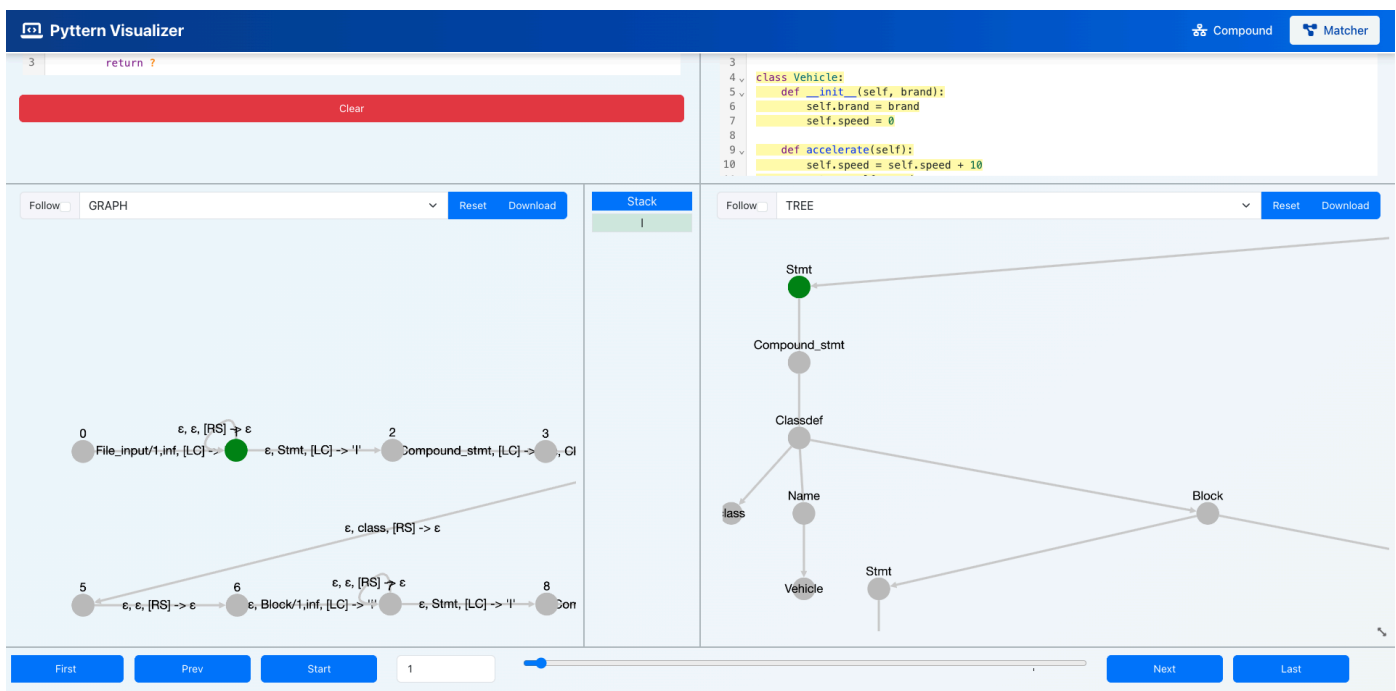


Figure 5.15 Matcher Page during Step Navigation With Graph, Tree and Stack Visible

This chapter presented the design and implementation of Pyttern Visualizer, covering the Compound page for batch analysis, result filtering and export, and the integration with the Matcher page for detailed step-by-step inspection. The following chapter examines how the implementation was validated through software testing and how the codebase is structured to support long-term maintainability.

CHAPTER 6

Software Validation and Maintainability

This chapter presents the validation strategy for the Pyttern Visualizer. The focus is on two complementary aspects. Section 6.1 describes the software engineering practices that make the codebase maintainable and extensible. Section 6.2 covers the automated testing approach that verifies correctness across different layers of the application. This chapter addresses only the technical validation of the software itself. Usability evaluation with end users is covered in the next chapter (Chapter 7).

6.1 Software Maintainability

Software maintainability refers to the ease with which a software system can be modified to fix defects, improve performance, or adapt to a changed environment [20]. In the context of React web applications specifically, maintainability challenges often stem from overly complex components, tightly coupled component hierarchies, and poor state management practices, all of which can increase technical complexity and reduce code quality over time [17]. A maintainable codebase allows developers to locate relevant code quickly, change one part of the system without breaking others, and extend functionality with minimal effort. This section describes the architectural decisions, the layered data flow design, the project structure, the reusable component library, and the refactoring choices that contribute to the maintainability of the Pyttern Visualizer.

Feature-Based Modular Architecture

The project structure follows a feature-based modular architecture. Each feature, namely compound and matcher, is a self-contained module with its own state management, business logic, API layer, and view components. The two features share no direct imports between each other. All shared code is located in the `common/` directory.

This separation ensures that changes in one feature do not affect the other, while all compound matching files are organized within a single directory.

The Redux documentation explicitly recommends this "feature folders" approach over the traditional Rails-style layout where actions, reducers, and components each occupy separate top-level directories [8]. In a Rails-style structure, tracing a single user action requires navigating across multiple folders. In a feature-based structure, the entire vertical slice from the Redux state definition to the API call to the React component lives in one place.

The Pyttern Visualizer adopts a hybrid of the feature-folder and Ducks/Slices conventions. The project is organized into feature folders, and within each feature the Redux slice file colocates actions and reducers in a single file. This combination ensures that all code related to a feature resides in a single directory, making features self-contained and independently navigable.

This consistent structure has practical consequences for maintainability. Adding a new feature requires creating a new folder under features/ with the same layered structure. No existing files need to be modified beyond registering the new slice in the store configuration. Therefore, investigating a bug requires navigating to only one feature folder, where everything from state management to UI rendering resides in one place.

Layered Data Flow Architecture

Within each feature module, the code follows a consistent six-layer data flow pattern that separates concerns into distinct, independently modifiable layers [14]. Figure 6.1 illustrates this architecture.

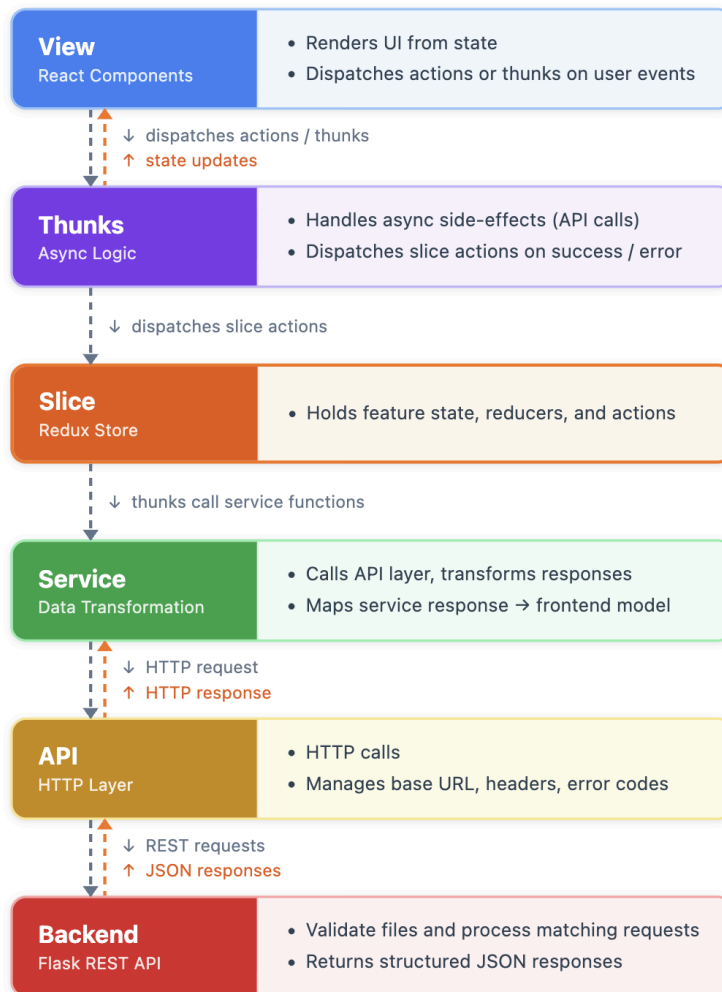


Figure 6.1 Layered Data Flow Architecture

Each layer has a single, well-defined responsibility. The View layer contains React components that render the UI from state and dispatch actions. They hold no business logic or HTTP calls, so changes to how a screen looks never affect how data is fetched or processed. The Thunks layer orchestrates asynchronous operations such as API calls, managing loading indicators and error notifications without transforming data itself. The Slice layer holds the Redux state, reducers, and actions, keeping all state transitions in one predictable place. The Service layer contains pure functions that handle data transformation and business logic, making them straightforward to test in isolation and easy to modify without touching the UI. The API layer is a thin HTTP abstraction so that if a backend endpoint changes, only one file needs to be updated. Finally, the Backend is the existing Pyttern Flask REST API that validates files and runs pattern matching.

Because each layer depends only on the layer directly below it, a change in one area does not ripple through the rest of the application. Changing how match results are sorted requires a modification only in the service layer, and a change in the backend API requires a modification only in the API layer. As a result, maintaining the application requires less effort with less complexity.

Reusable Components

The `common/components/` directory contains seven shared UI components that are used across both features. Rather than duplicating UI logic within each feature, commonly needed features were extracted into configurable components.

The `ExpandableCard` handles collapsible content sections and is used across both features wherever content needs to be revealed on demand. The `CodeView` provides a read-only, syntax-highlighted code display with optional error line highlighting, used consistently in both code file cards and pattern file detail modals. The `ValidationErrorCard` renders validation error details in a uniform format, regardless of whether the error comes from a code file or a pattern file. The `ConfirmationModal` guards destructive operations throughout the Compound page and navigation flow. The `InfoModal` provides contextual help dialogs, and the `AppLoader` displays a global loading overlay tied to the Redux store.

The same principle applies at the feature level. Each feature's `view/` directory has its own `components/` subdirectory for sub-components that are reused within that feature but are not relevant to the rest of the application. The `PatternTree` component is a good example. It is used in two places within the Compound feature, once to display the imported compound pattern and once inside each code file card to show the per-file annotated match results as mentioned in ‘Code Files Section’ sub-section of Section 5.3. Because it is specific to the Compound feature, it lives in the feature's own `view/components/` folder rather than in `common/`. This two-level component organisation keeps the `common/` directory focused on genuinely cross-feature concerns while still avoiding duplication within a feature.

The maintainability benefit is straightforward. Any change to how validation errors are displayed, for instance, requires updating only `ValidationErrorCard` and is reflected everywhere in the application. Without these shared components, the same fix would need to be applied in multiple places independently.

Additionally, each component has its own CSS file located in its directory, keeping styling isolated. This structure means that modifying the appearance of one component does not affect others.

Type Safety with TypeScript

The application uses TypeScript throughout the codebase. Dedicated model files define interfaces and enums for all data structures, including compound patterns, code files, validation errors, match results, and filter configurations. This type coverage catches structural errors at compile time rather than at runtime. For example, if a developer accidentally passes a validation error object where a match result is expected, the TypeScript compiler reports the mistake before the code runs. From a maintainability perspective, this means the codebase can evolve with confidence that type inconsistencies will be caught before they reach the user, making future changes less risky.

Centralized Error Handling

Error handling follows a multi-level strategy that ensures consistent user feedback while preserving debugging information.

At the API layer, HTTP response status codes are translated into user-readable error messages. Network failures are detected and mapped to a network error message. A shared utility function `getApiErrorMessage()` normalises error objects of various types into a consistent string format, ensuring that error messages are always presentable to the user.

At the thunk layer, all asynchronous operations are wrapped in try-catch blocks. Errors are displayed to the user through toast notifications and updating the state by the Redux slice layer.

At the Redux slice layer, each slice defines rejected cases for its asynchronous thunks, updating the state to reflect the error condition so that the UI can respond appropriately by disabling buttons or showing error badges.

This layered error handling means that no error disappears silently. Every error is caught at the appropriate level, presented to the user in a readable form, and recorded in the application state so that the UI remains consistent.

Refactoring Decisions

Throughout the development of the Pyttern Visualizer, several refactoring decisions were made to reduce complexity, eliminate duplication, and improve the testability of the codebase.

Extraction of domain-specific utilities into dedicated modules. As the Compound feature grew, several concerns initially embedded in components or thunks were extracted into standalone utility modules. The `importUtils.ts` module encapsulates all file import and structural validation logic. By isolating these concerns from both the UI layer and the Redux state management, each function can be unit-tested with plain JavaScript objects without requiring browser APIs or a running Redux store. Similarly, `csvUtils.ts` encapsulates the CSV export logic, separating generation from download so that the CSV content can be tested without triggering browser side effects and the `langUtils.ts` module centralises language detection and extension mapping, ensuring that adding support for a new language requires a change in exactly one place.

Separation of service logic from Redux thunks. Early in development, data transformation logic was performed directly inside thunks. This included mapping API responses to the application's data model, sorting match results, and propagating flags through the pattern tree. This was refactored so that thunks serve only as orchestrators while all data transformation is processed in the service layer as pure functions. This separation means that service functions can be tested by passing plain objects and asserting on return values, without mocking Redux dispatch or the global store.

Recursive tree operations as reusable service functions. The compound pattern tree model requires several recursive traversal operations, including finding a pattern file by filename, extracting all leaf pattern files, toggling selection state across the tree, and updating pattern files with validation or match results. Rather than implementing these traversals inline within components or slice reducers, they were consolidated in `compoundService.ts` as standalone functions. Each function uses a type guard to distinguish leaf nodes from branch nodes during traversal. This ensures that the tree traversal logic is defined once, tested thoroughly, and reused consistently across validation, matching, filtering, and UI rendering.

6.2 Software Testing

This section describes the testing strategy, the tools and methodology used, the role of AI-assisted test data generation, and the results of the test suite.

Testing Framework and Configuration

The application uses Vitest [19] as its testing framework, chosen for its native integration with the Vite build tool [18] and its compatibility with TypeScript. The test configuration enables global test utilities, eliminating the need for explicit imports in every file. Test file patterns cover both unit tests located alongside source files and integration tests in a dedicated `tests/` directory.

Unit Tests

Unit tests are located in `__tests__` directories within each feature module and the common utilities. The unit test suite covers the following layers of the application.

Utility function tests verify the language detection and extension mapping functions in `langUtils.ts`. These tests cover standard cases as well as edge cases such as unknown extensions, empty strings, and filenames with multiple dots. There are 8 tests in this category.

Redux slice tests verify the state management behavior for both features and the shared loader. The `appLoaderSlice` tests confirm that the loading state works correctly. The `matcherSlice` tests cover all reducer actions for the `Matcher` feature, including pattern and code setting, validation error management, match lifecycle transitions, step navigation, and full state reset. The `compoundSlice` tests also cover all reducer actions for the `Compound` feature, including code file loading, compound pattern setting, pattern selection and filter management, state reset, and the extra reducers that handle asynchronous thunk lifecycle states. Together these amount to 37 tests across 3 files.

Service function tests verify the business logic and data transformation layer. The `matcherService` tests cover the API-response mapping, including the column-offset adjustment applied to code positions and error message formatting for various input types. The `compoundService` tests cover tree traversal functions at various nesting depths, recursive pattern selection for the filtering feature, and getting all pattern files of a compound pattern. There are 27 tests across 2 files.

Import and validation utility tests verify that compound patterns are correctly recognised and structurally valid before any matching begins. This includes checking that operator names are valid, that NOT operators have exactly one child, that no operator is left empty, and that pattern files are not empty. Edge cases such as errors in deeply nested trees are also covered. There are 17 tests in this category.

CSV generation tests check correct header generation, alphabetical pattern column sorting, status representation for different match outcomes and handling of files with missing results. There are 8 tests in this category.

Integration Tests

The integration test suite validates the complete compound matching pipeline against the real Pyttern backend. Unlike unit tests, which mock external dependencies, these tests execute actual HTTP requests to the backend. The test suite uses 7 compound patterns containing a total of 11 pattern files, with 5 to 6 code files per compound pattern. How the test data was gathered is explained in the next sub-section.

Inside the integration tests, for each of the 7 compound patterns, three test cases are executed. The first test validates all code files associated with the pattern against the backend and asserts that every file receives validated status with no errors. The second test validates all pattern files in the compound pattern tree against the backend. The third test executes the full workflow of pattern validation, code file validation, and compound matching, then verifies the match status. It also verifies that the result list is correctly sorted with matched files before not-matched files, and that every code file receives an individual match outcome for each leaf pattern in the compound tree, not just an overall matched or not-matched verdict. There are 21 integration tests in this category.

Test Data and AI-Assisted Generation

The test data resides in a dedicated Test-Data/ directory and comprises two categories.

The first category consists of compound pattern files. There are 7 compound patterns containing a total of 11 .pyt pattern files as mentioned in the ‘Integration Tests’ sub-section. These patterns were provided by the Pyttern development team and represent real-world code analysis scenarios such as detecting debug print statements without return values, unused loop variables, parameter erasure, and improper `__init__` implementations.

The second category consists of example code files. There are 39 Python source files distributed across the 7 patterns, with 5 to 6 files per compound pattern. These code files were generated using Claude Code with a custom-authored skill named Pyttern. In Claude Code, a skill is a locally defined instruction file that extends the AI assistant with domain-specific knowledge and task guidelines, following a structured format that Claude Code recognizes and loads automatically [21].

The skill encoded the Pyttern domain knowledge that the AI would otherwise lack, such as the wildcard syntax, the boolean operators used to build compound patterns, and worked examples of the seven compound patterns. With this knowledge the AI could generate code that is semantically meaningful for each pattern's logic, rather than superficially plausible Python that does not exercise the intended matching behavior. Each generated file also carries a header comment recording the target compound pattern and its expected per-sub-pattern and overall outcomes, which kept the test data easy to audit by hand and consistent across all seven patterns.

The filename convention uses a `match_` prefix for files expected to match and a `no_match_` prefix for files expected not to match. This convention enables the integration tests to automatically verify correctness without hardcoded expected-result mappings.

This approach demonstrates a practical application of AI-assisted test data generation. The compound patterns define what to detect, the skill provides the AI with how the detection works, and the AI generates diverse code examples that exercise both positive and negative matching paths across all seven compound patterns.

Tests Summary

Table 6.1 provides an overview for the tests across the application.

Category	Files	Test Cases	Scope
Utility Functions	1	8	Language detection and extension mapping
Redux Slices	3	37	State management for both features and shared loader
Service Functions	2	27	Business logic, tree traversal, error formatting
Import and Validation Utilities	1	17	Pattern structure validation, operator recognition
CSV Generation	1	8	Export formatting, escaping, edge cases
Integration Tests	1	21	Full pipeline with validation, matching, and result verification for the Compound section
Total	9	123	

Table 6.1 Tests Overview

Tests Coverage

To quantify how thoroughly the test suite exercises the application, code coverage was measured using Vitest's built-in V8 coverage provider.

As described in Sections 6.1 and 6.2, the testing strategy deliberately targets the business-logic layer of the layered data flow architecture, the Redux slices, service functions, and domain-specific utilities. The view layer is intentionally excluded from unit testing because React components are validated through the user evaluation reported in Chapter 7, while the thunk layer and its HTTP API wrappers are exercised end-to-end by the integration test suite against the real Pyttern backend.

The business-logic modules reach strong line coverage, with files falling between 75% and 100%. The overall project-wide line coverage is 25.21%, which is not high but expected. The majority of uncovered code belongs to the React view components, which are intentionally excluded from unit testing and evaluated through the user study in Chapter 7. The thunk layer is similarly validated through the integration tests against the real backend. Overall, the coverage results are consistent with the testing strategy, and the layers that matter most for correctness are well covered.

Test Results

All unit tests and integration tests pass consistently confirming that both the client-side logic and integration with backend are correct. These results validate both the frontend's data handling logic and its correct integration with the Pyttern backend API. The test results are shown in Figure 6.2.

```
✓ src/features/matcher/__tests__/matcherSlice.test.ts (13 tests) 8ms
✓ src/features/compound/__tests__/compoundService.test.ts (16 tests) 94ms
✓ src/features/compound/__tests__/compoundSlice.test.ts (20 tests) 12ms
✓ src/common/slices/__tests__/appLoaderSlice.test.ts (4 tests) 4ms
✓ src/features/matcher/__tests__/matcherService.test.ts (9 tests) 5ms
✓ src/features/compound/__tests__/importUtils.test.ts (17 tests) 4ms
✓ src/common/utils/__tests__/langUtils.test.ts (13 tests) 4ms
✓ src/features/compound/__tests__/csvUtils.test.ts (8 tests) 4ms
✓ tests/integration/compound/compound-section.test.ts (21 tests) 322ms

Test Files  9 passed (9)
  Tests    121 passed (121)
Start at   23:53:49
Duration   710ms (transform 499ms, setup 0ms, import 812ms, tests 457ms, environment 1ms)
```

Figure 6.2 Test Results in the Local Environment

This chapter presented the technical validation of the Pyttern Visualizer, covering the architectural and coding practices that support long-term maintainability and the automated test suite that verifies the correctness of the application. All 123 unit and integration tests pass consistently, and the business-logic modules reach line coverage between 75% and 100%. The following chapter evaluates the tool from the perspective of end users through a structured usability study.

CHAPTER 7

User Validation

After the technical validation reported in Chapter 6, the Pyttern Visualizer was evaluated from the perspective of end users through a structured usability study. The aim was to confirm that the design decisions described in Chapter 5, in particular the compound pattern view, the multi-file overview, and the filtering interface, support the teaching-oriented workflow that motivated this thesis. This chapter explains how the study was conducted, presents the collected quantitative and qualitative results, discusses what these results imply for the strengths and weaknesses of the tool, and finally outlines the limitations that affect how broadly the findings can be generalised.

7.1 Method

Goals of the Validation

The goal of the validation study was to assess whether the extended Pyttern Visualizer is usable by participants with a programming background and whether the new Compound page actually delivers the teacher-oriented workflow that the thesis aims to provide. The study was designed to answer three concrete questions. The first one was whether participants could complete realistic pattern-matching tasks with the tool without external help. The second one was how participants perceived the usability of the system as measured by the System Usability Scale [5]. The third one was whether the specific workflow features introduced in this thesis, such as the compound pattern tree, the filtering controls, and the visual distinction between ‘matched’, ‘not matched’, and ‘not validated’ files, were perceived as effective.

Participants

Fifteen participants took part in the test. All of them had a programming background and were students of the Programming Paradigms course (LINFO2335) at UCLouvain. This course was chosen for two reasons. First, its students are Master's-level Computer

Science students who are already fluent in multiple programming languages and paradigms. Second, the course content directly involves comparing different programming paradigms, and reasoning about how programs are structured. This aligns closely with what the usability test asked participants to do, making them reasonable candidates for the usability test.

Test Plan

The session was organized into four phases. It began with a short introduction and a demo in which the moderator explained what Pyttern is, including patterns, compound patterns and walked the participants through a minimal example. Participants were then asked to play the role of teachers grading student submissions.

Task A asked them to build a basic compound pattern named `ProperClassInit`, which checks that a submitted Python class correctly defines an `__init__` method that does not return a value while also having at least one other method that does return a value. The participants received three primitive Pyttern pattern files: `has_class_init.pyt`, which detects whether a class defines an `__init__` method; `has_method_return.pyt`, which detects whether a class contains at least one method with a return statement; and `has_init_return.pyt`, which detects whether `__init__` itself returns a value. They had to combine the patterns into a compound one that checks whether a class has an `__init__` method, has at least one method returning a value, and does not return a value from `__init__`. Once the compound pattern was built, they uploaded the five student code files, ran the match, and answered six questions about the expected result, such as which files matched, which did not, which failed validation, and the cause of the validation error.

Task B introduced a more complex compound pattern named `ProperInheritance`, which checks that a submitted Python class correctly inherits from a parent, calls `super().__init__()` in its constructor, does not return a value from `__init__`, and has at least one method returning a value. It was built from five pattern files and applied to twelve student code files. As shown in Figure 7.1, the compound pattern combined parent-child inheritance, a call to `super().__init__()`, the absence of a return inside `__init__`, and the presence of at least one method returning a value either directly or inside a conditional branch.

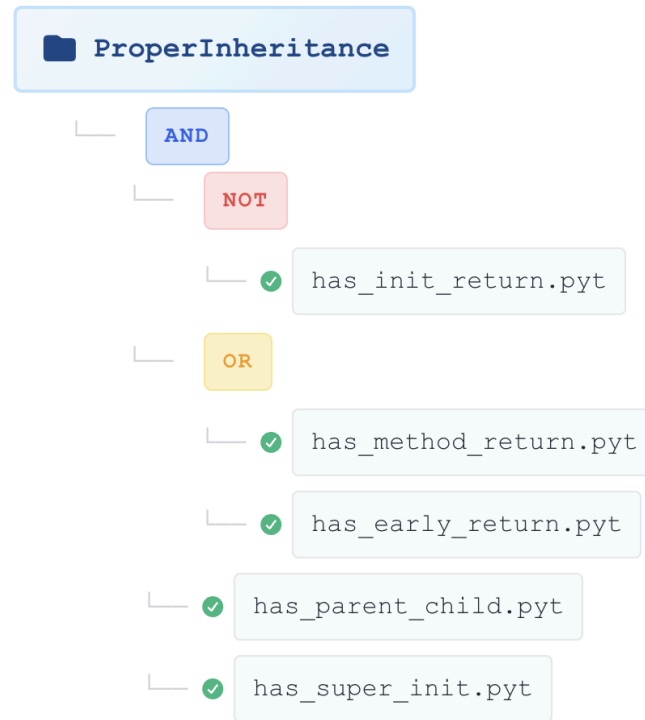


Figure 7.1 The Compound Pattern of Task B

The questionnaire itself was delivered through three INGINIOUS tasks named Pyttern-Visualizer (Task A), Pyttern-Visualizer (Task B), and Pyttern-Visualizer (Feedbacks). The first two tasks collected the participants' answers to the task-specific questions, such as which files matched, which ones did not, and which ones could not be validated. The third task collected the post-test questionnaire, namely the SUS items, the workflow items, and the open-ended feedback.

Metrics and Data Collection

The post-test questionnaire combined three instruments. The first one was the System Usability Scale [5], a standard ten-item questionnaire alternating positive-worded and negative-worded statements on a five-point likert scale, scored on a single scale from 0 to 100, where a higher score means better perceived usability. Because the questionnaire mixes positive-worded and negative-worded statements, the scoring procedure first adjusts each answer so that a higher value always reflects a better judgement, and it then combines the ten adjusted answers into the final score following the standard procedure defined by the scale. The second instrument was a set of four workflow-specific questions targeted at the new features introduced by this thesis, namely the readability of the compound pattern

tree, the usefulness of the filter, the visual distinction between matched, not matched, and not validated states, and the perceived usefulness of the tool in a realistic teaching scenario. The third instrument was an open-ended question that invited participants to describe any difficulties they encountered and any features they would like to see.

Setup

The test environment was distributed as a self-contained Docker package. Each participant received a setup folder with a README, two Compose files for the supported CPU architectures (amd64 and arm64), and the code and pattern files needed for the two tasks. Starting the application required only Docker Desktop and a single Docker Compose command. This setup avoided installation differences across operating systems and made sure that every participant tested the same version of the tool without dependency problems. One participant was unable to start the application with the Docker Compose command on his local machine due to local problems with Docker. For this case the frontend and backend repositories were cloned and run locally instead, which allowed the participant to complete the usability test successfully.

7.2 Results and Discussion

System Usability Scale

The average SUS score across the fifteen participants was 79.0, with a median of 80.0 and a standard deviation of about 15.5. Scores ranged from 42.5 to 100.0. The adjective rating scale is a common way to interpret SUS results, since a score on its own is hard to judge. It maps a numeric score onto a descriptive label [22]. An average of 79 sits in the 'Good' band of this scale, which covers scores from roughly 70 to 80. Eleven of the fifteen participants placed the tool in the Good or Excellent bands, two of them gave OK scores, and only two gave clearly lower ratings (S04 with 42.5 and S11 with 55.0). More than two thirds of the participants therefore rated the tool at Good or above, which reaches the level of satisfaction associated with usable interactive systems.

SUS - Answers

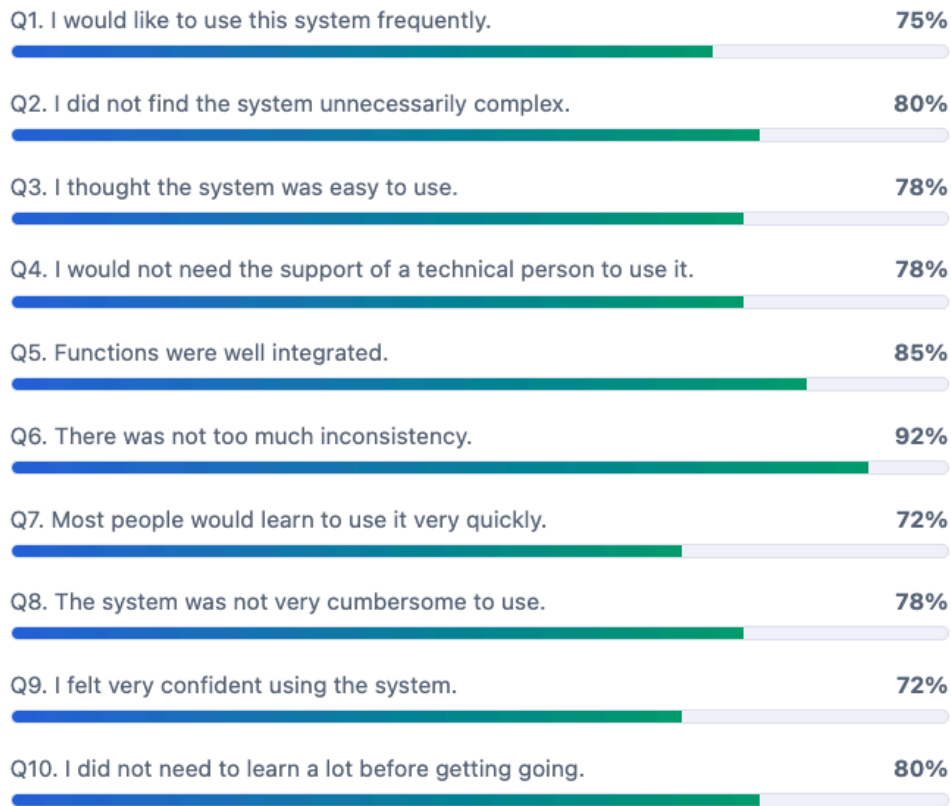


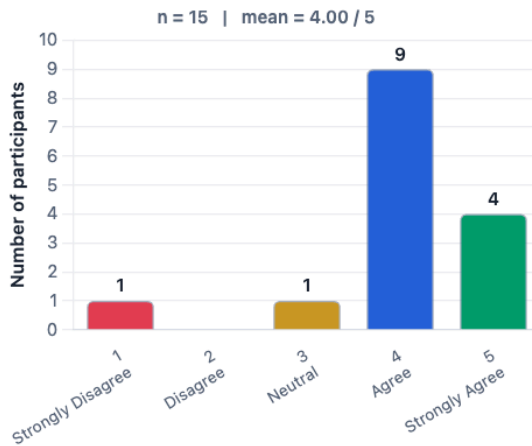
Figure 7.2 SUS Agreement Percentages Per-Item

The per-item view in Figure 7.2 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%). This score is still positive, but it points to the learning curve of the compound pattern model as the dimension most worth improving.

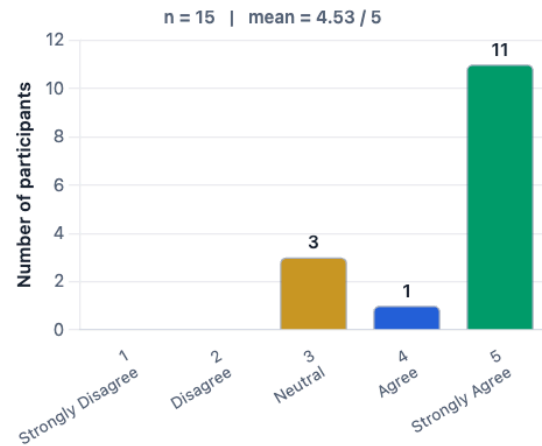
Workflow-Specific Questions

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 7.3.

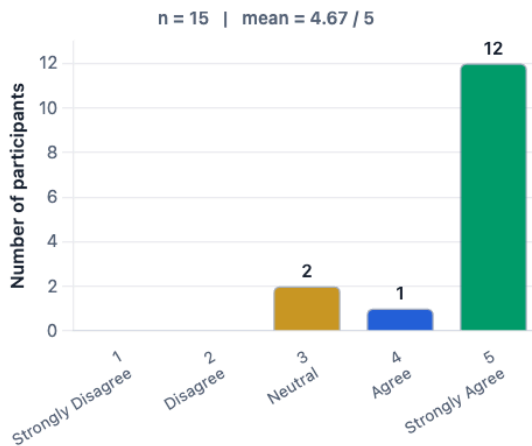
Q11 — Compound pattern structure was easy to understand



Q12 — The filter feature helped find results efficiently



Q13 — Matched / not-matched / not-validated were clearly distinguishable



Q14 — Tool would be useful in a real teaching scenario

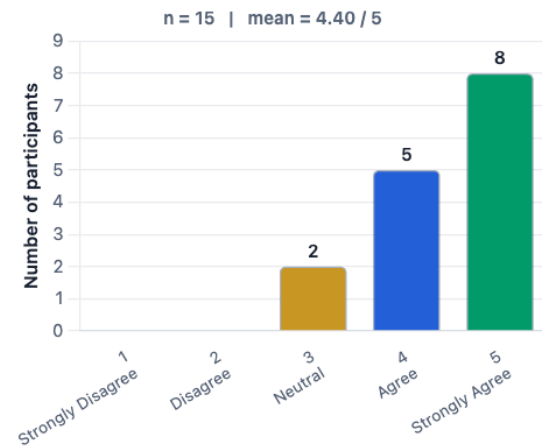


Figure 7.3 Workflow-Specific Question Answer Distributions

The strongest result was Q13, which shows that participants could reliably distinguish between matched, not matched, and not validated files. This validates the choice to combine color and shape as redundant encodings of file status. The same redundancy made the tool usable for a red-green colorblind participant (S09), who reported that the shapes complemented the color cues well, which is a concrete accessibility outcome.

The filter (Q12) was widely seen as efficient for narrowing down the visible files. This supports the design choices presented in Chapter 5 that follow the filtering guidelines of Bjäräng and Enderstein [12]. Q14 asked whether the tool would be useful in a real teaching scenario, the question that targets the central motivation of this thesis. Even

participants who reported difficulties in the open feedback still rated the tool as useful for teaching, which suggests that the limitations they raised are perceived as polish items rather than fundamental barriers.

The compound-pattern question Q11, on whether the AND/OR/NOT tree was easy to understand, averaged 4.00 and had the widest spread of the four items. Thirteen of the fifteen participants placed it on the agreement side. The single Strongly Disagree answer came from S04, who also produced the lowest SUS score (42.5) and the shortest comment, which simply asked for an introduction and a bigger example. The negative signal is therefore concentrated in the participant who struggled most with the test as a whole. The comments from S06, S14, and S15 describe a learnability difficulty around the operators rather than a conceptual barrier. S06 wrote that they "had a few difficulties understanding how to use the AND/NOT/OR operators in the tree file" and added that the README in the main Pyttern GitHub repository resolved the confusion. S14 asked for "a bit more information about how we can implement the rules (AND, OR, NOT)". S15 reported that the "only difficulty was understanding the correct folder structure for AND, OR, and NOT patterns". The three remarks converge on the same operator semantics gap, which suggests that brief in-app guidance would close it.

Open-Ended Feedback

Fourteen of the fifteen participants provided written feedback, and nine of them explicitly praised the tool. They described the interface as clean, pleasant, well-designed, and impressive, with S02 writing that "the interface looks really good, it's pleasant to use" and S05 stating that "overall the structure is really solid and well-designed". This consistent praise supports the quantitative evidence that the overall design works.

The most frequent improvement suggestion targeted the current requirement to construct the compound pattern outside the tool by creating folders. Five participants (S03, S08, S10, S11, S15) asked for an in-app builder that would let them assemble the AND/OR/NOT tree visually, for example through drag and drop, rather than by editing a folder structure outside the application. This is the dominant friction point in the study, and it is also visible in the wider spread of Q11. In the current version of the tool, the compound pattern is supplied as a folder structure prepared by the user outside the application. This construction step is therefore not part of the visualization itself, although in hindsight it could have been integrated into the tool, and it clearly shapes the overall user experience of the tool.

Three participants (S06, S14, S15) also reported that the operator semantics were confusing at first and looked for additional documentation. Two participants (S01, S05) asked for clearer validation error messages in the code view, though this concerns the Pyttern engine in the backend rather than the visualizer. Smaller suggestions included the option to add more code files after a match has been run without resetting the page (S02), a per-file summary table showing which individual patterns matched and which failed for each code file (S15), a history of previously imported patterns (S09), and reduced padding around components, since heavy padding and aggressive text truncation made the tool harder to use in split-screen mode on small displays such as 13-inch laptops (S01).

Overall Reading of the Results

The SUS scores, the scores of workflow-based questions, and the open-ended feedback all point in the same direction. The design successfully extends the Pyttern Visualizer by letting users work with several code files and compound patterns in one view, the visual encoding of match outcomes is clear, the filter is effective, and the tool is perceived as useful for teaching. The most notable improvement area concerns the construction of compound patterns, which could be addressed by adding an in-app builder. This direction is recorded in Chapter 9 (Future Work).

7.3 Threats to Validity

Sample Size

The study involved fifteen participants, which is an acceptable sample size for usability studies and sufficient to surface some relevant usability issues. However, the small number limits how far the SUS and workflow scores can be generalised. The reported averages should be read as rough indicators of user satisfaction rather than precise estimates, and a single outlying response can noticeably shift the mean, as seen in Figure 7.2 with S04 on Q11.

Participant Bias

All participants were students of the LINFO2335 course and not actual teachers. They have a programming background and are familiar with reasoning about code, so they are reasonable candidates for the target audience, but they are not the end users described in Chapter 1. Their evaluation of Q14, where they were asked to imagine themselves in the

role of teachers, is therefore an informed judgement rather than the lived experience of teachers grading their own course. Furthermore, the participants knew that the tool was being evaluated as part of a master thesis, which may have introduced a small positive bias in the questionnaire answers. The use of an anonymous online form and the inclusion of negative-worded SUS items reduce this effect.

This chapter presented the user validation of the Pyttern Visualizer, covering the method, the quantitative and qualitative results, and the threats to validity. The fifteen-participant usability study produced a System Usability Scale score of 79.0 and a workflow-question average of 4.40, supporting most of the design choices presented in Chapter 5 while also surfacing a set of improvement suggestions through the open-ended feedback. The following chapter draws the overall conclusions of the thesis.

CHAPTER 8

Conclusion

This thesis addressed a gap between what the Pyttern matching engine can do and what teachers can effectively use in a course. The engine already supports compound patterns and many files, but the existing visualizer was a debugging tool built around a single pattern and a single file, which left teachers with a fragmented and manual workflow. The bottleneck was in the visualization and interaction layer rather than in the language or the matcher. The solution is a frontend-only extension of the existing React application, the Compound page, which lets teachers import a compound pattern, match several code files in one action, inspect the outcome per file and per sub-pattern, filter and export the results, and open any pair in the existing Matcher view. Alongside this feature, the existing frontend project was refactored into a more modular and maintainable architecture so that further work can build on it with less effort.

The evaluation supports the central claim. An automated suite of 123 tests passes consistently with strong coverage of the business logic, and the usability study with fifteen participants produced an average System Usability Scale score of 79.0 together with a workflow-question average of 4.40 on a five-point scale. The visual encoding of file status and the filter were the strongest results, while the learnability of the compound pattern model was the weakest. In practical terms, the tool removes much of the manual effort that Pyttern previously demanded, since the batch match replaces the repeated single matches and external bookkeeping of the results.

The findings come with limitations. The fifteen participants are enough to surface most usability issues but too few for broadly generalisable scores, and they were students rather than practising teachers, so their judgement of the teaching scenario is informed rather than lived. The most frequent request concerned building the compound pattern outside the tool as a folder, which points to an in-app builder as a natural next step. This and other directions are taken up in the following chapter. Overall, the extended Pyttern Visualizer closes the workflow gap that motivated this thesis and turns Pyttern into a tool that supports teacher-oriented workflows.

CHAPTER 9

Future Work

The extended Pyttern Visualizer covers the core workflow that motivated this thesis, and the evaluation pointed to three directions that would make it more complete.

The clearest is a built-in compound pattern builder. The tool currently receives the compound pattern as a folder prepared outside the application and only displays the resulting AND/OR/NOT tree. This was the most frequent request in the usability study, raised by five participants. An in-app builder would let teachers assemble the logical tree directly, for example through drag and drop, which would reduce the learnability hurdle around the operators and remove the need to manage folders by hand.

A second direction is to embed the visualizer into INGIInious, the autograder used to grade assignments in the target courses. The tool is currently a standalone application that teachers open separately from where they grade submissions. Integrating the React application into the grading platform would place pattern detection inside the workflow teachers already use, so that the structural feedback from Pyttern can be inspected next to the rest of a submission.

A third direction is more advanced filtering. The current filter narrows files by match status and by selected pattern nodes, which participants found efficient. It could be extended with combined conditions across sub-patterns, free-text search over file names, and a per-file summary table of pattern outcomes. These additions would help teachers work with larger sets of submissions, where a simple status filter is less sufficient on its own.

Bibliography

- [1] G. Haldeman, A. Tjang, M. Babeş-Vroman, S. Bartos, J. Shah, D. Yucht, and T. D. Nguyen, "Providing meaningful feedback for autograding of programming assignments," in Proc. 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18), Baltimore, MD, USA, Feb. 2018, pp. 278–283, doi: 10.1145/3159450.3159502.
- [2] J. Liénard, K. Mens, and S. Nijssen, "Pyttern: a Python-based program query language," in Proc. 23rd Belgium-Netherlands Software Evolution Workshop (BENEVOL 2024), Namur, Belgium, Nov. 21–22, 2024, vol. 3941, CEUR Workshop Proceedings, pp. 88–96.
[Online]. Available: https://ceur-ws.org/Vol-3941/BENEVOL2024_TECH_paper10.pdf
- [3] J. M. Carroll, "Human-computer interaction: Psychology as a science of design," International Journal of Human-Computer Studies, vol. 46, no. 4, pp. 501–522, Apr. 1997, Academic Press, doi: 10.1006/ijhc.1996.0101.
- [4] S. P. Yatam, "Evaluation and redesign of an interactive system by applying HCI principles," B.S. thesis, Faculty of Computing, Blekinge Institute of Technology, Karlskrona, Sweden, May 2021.
- [5] W. S. L. Nasution and P. Nusa, "UI/UX design web-based learning application using design thinking method," ARRUS Journal of Engineering and Technology, vol. 1, no. 1, pp. 18–27, 2021, doi: 10.35877/jetech532.
- [6] C. Adhitya, R. Andreswari, and P. F. Alam, "Analysis and design of UI and UX web-based application in Maiproyek startup using user centered design method in information system program of Telkom University," IOP Conference Series: Materials Science and Engineering, vol. 1077, no. 1, art. no. 012039, 2021, IOP Publishing, doi: 10.1088/1757-899X/1077/1/012039.
- [7] "React: The library for web and native user interfaces," React Official Documentation. Accessed: May 18, 2026. [Online]. Available: <https://react.dev/reference/react>

- [8] "Redux: A predictable state container for JS apps," Redux Official Documentation. Accessed: May 18, 2026. [Online]. Available: <https://redux.js.org/>
- [9] S. Narciss, "Feedback strategies for interactive learning tasks," in *Handbook of Research on Educational Communications and Technology*, 3rd ed., J. M. Spector, M. D. Merrill, J. J. G. van Merriënboer, and M. P. Driscoll, Eds., 2008, pp. 125–144.
- [10] H. Keuning, "Automated feedback for learning code refactoring," Ph.D. dissertation, Open Universiteit, Heerlen, The Netherlands, 2020. [Online]. Available: <https://research.ou.nl/en/publications/automated-feedback-for-learning-code-refactoring>
- [11] A. Lewandowska and A. Olejnik-Krugly, "Do Background Colors Have an Impact on Preferences and Catch the Attention of Users?," *Applied Sciences*, vol. 12, no. 1, art. no. 225, Dec. 2021, MDPI, doi: 10.3390/app12010225.
- [12] K. Bjäräng and V. Enderstein, "Designing filtering for an analytic software," M.S. thesis, Dept. of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden, Rep. CSE 22-104, 2022.
- [13] M. Busch and N. Koch, "Rich Internet Applications: State-of-the-art," Institut für Informatik, Ludwig-Maximilians-Universität München, Munich, Germany, Tech. Rep. 0902, Dec. 2009.
- [14] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley, 2002. ISBN: 978-0-321-12742-6.
- [15] M. Madsen, O. Lhoták, and F. Tip, "A semantics for the essence of React," in *Proc. 34th European Conference on Object-Oriented Programming (ECOOP 2020)*, LIPIcs, vol. 166, pp. 12:1–12:26, Dagstuhl, Germany: Schloss Dagstuhl Leibniz-Zentrum für Informatik, 2020, doi: 10.4230/LIPIcs.ECOOP.2020.12.
- [16] D. Pronina and I. Kyrychenko, "Comparison of Redux and React Hooks methods in terms of performance," in *Proc. 6th International Conference on Computational Linguistics and Intelligent Systems (COLINS 2022)*, Gliwice, Poland, May 12–13, 2022, vol. 3171, CEUR Workshop Proceedings, pp. 791–800. [Online]. Available: <https://ceur-ws.org/Vol-3171/paper59.pdf>

- [17] P. Bollu, "Ensuring maintainability in React web applications," M.S. thesis, Tampere University, Tampere, Finland, 2024. [Online]. Available: <https://trepo.tuni.fi/bitstream/10024/161820/2/BolluPrathibha.pdf>
- [18] "Vite: Next generation frontend tooling," Vite Official Documentation. Accessed: May 18, 2026. [Online]. Available: <https://vite.dev/guide/>
- [19] "Vitest: A Vite-native test framework," Vitest Official Documentation. Accessed: May 18, 2026. [Online]. Available: <https://vitest.dev/guide/>
- [20] IEEE, IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990, New York, NY: IEEE, 1990.
- [21] "Claude Code: Skills," Claude Code Official Documentation. Accessed: May 18, 2026. [Online]. Available: <https://code.claude.com/docs/en/skills>
- [22] A. Bangor, P. Kortum, and J. Miller, "Determining what individual SUS scores mean: Adding an adjective rating scale," *Journal of Usability Studies*, vol. 4, no. 3, pp. 114–123, May 2009.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl