Università
della
Svizzera
italiana

Faculty
of Informatics

Bachelor Thesis

July 1, 2026

From Raw Data to Pedagogical Insights: An Interactive Dashboard Leveraging Static Analysis for PyTamaro-Web

Simone Maccario

Abstract

Interactive educational platforms like PyTamaro-Web provide hands-on programming instruction. Within this context, instructors act as authors, designing specific coding activities for users to solve by progressively writing and executing code in their browsers. While this constant process of executing code encourages active learning, it also produces large quantities of raw execution telemetry. However, authors face a significant pedagogical blind spot: they lack a systematic way to visualize this data in order to understand how users complete the tasks or where frequent static errors occur. In order to extract meaningful insights and avoid superficial or trivial analysis, it is necessary to reconstruct anonymous user sessions by combining relational database execution records with external file system assets and executing computationally expensive static analysis on thousands of code submissions.

This thesis proposes an analytics pipeline that generates the Interactive PyTamaro-Web Dashboard, which transforms raw educational data into useful teaching insights. The solution provides a smooth integration of heavy data processing and user-friendly visualization. The backend architecture combines PostgreSQL code execution data with Jupyter Notebook templates efficiently, running the `ty` static analyzer at scale using an optimized batch-processing engine with global caching. The pipeline then outputs a highly portable, offline-ready HTML interface that operates entirely without a web server. This static dashboard enables authors to explore curricula and user execution timelines dynamically, offering interactive side-by-side code diffs and detailed syntactic and semantic error highlighting.

In order to ensure that the tool meets real-world teaching needs, an in-depth usability study was conducted with an instructor to evaluate the system. This assessment covered the dashboard's pedagogical effectiveness, visual clarity and ease of use. The feedback confirmed that the system successfully breaks down complex execution histories into straightforward visual timelines. By reconstructing user sessions and extracting metrics on the most frequent errors, the dashboard provides instructors with a clearer understanding of how users approach programming tasks and where they encounter difficulties. Overall, the tool enables authors to assess the effectiveness of their activities, identify recurring challenges, and gain actionable insights into the learning process.

Advisor
Prof. Matthias Hauswirth
Assistants
Joey Bevilacqua, Dr. Luca Chiodini

Advisor's approval (Prof. Matthias Hauswirth): Date:

Contents

1 Introduction **2**
- 1.1 Key Technical Contributions . . . 2
- 1.2 Structure of the Thesis . . . 2

2 Background and Problem Statement **3**
- 2.1 Related Work in Educational Telemetry . . . 3
- 2.2 The PyTamaro-Web Educational Environment . . . 3
- 2.3 Dual-Storage Platform Architecture . . . 4
  - 2.3.1 File System Content and Instructor Repositories . . . 5
  - 2.3.2 Relational Telemetry and Anonymous Event Logging . . . 5
- 2.4 The Instructor's Blind Spot: Bridging the Feedback Gap . . . 6
- 2.5 Technical Challenges . . . 6
  - 2.5.1 Reconstructing User Sessions . . . 6
  - 2.5.2 Scalability of Static Code Analysis . . . 7
  - 2.5.3 Constraints of an Offline Interactive Dashboard . . . 7

3 Designing the Analytics Pipeline and Dashboard **8**
- 3.1 High-Level System Architecture . . . 8
- 3.2 CLI-Based Application Initialization . . . 8
- 3.3 Database-Related Optimizations . . . 9
- 3.4 Object-Oriented Data Modeling . . . 9
- 3.5 Data Builder and Object Hydration . . . 11
  - 3.5.1 Activity and Variant Reconstruction . . . 11
  - 3.5.2 Curriculum Aggregation and Metric Extraction . . . 11
- 3.6 Optimized Static Analysis and Global Caching . . . 11
- 3.7 Static Frontend Architecture and Visualizations . . . 12
  - 3.7.1 Author List . . . 13
  - 3.7.2 Author Dashboard: Activities and Curricula . . . 14
  - 3.7.3 Curriculum Page . . . 15
  - 3.7.4 Activity Page . . . 16

4 System Evaluation **20**
- 4.1 Evaluation Methodology . . . 20
  - 4.1.1 Dashboard-Wide Tasks . . . 20
  - 4.1.2 In-Depth Activity Tasks . . . 20
- 4.2 Survey Design and Metrics . . . 20
- 4.3 Results and Discussion . . . 21
  - 4.3.1 System Usability Scale (SUS) Results . . . 21
  - 4.3.2 Pedagogical Insights and Value . . . 21
  - 4.3.3 Constructive Feedback and Future Improvements . . . 21

5 Conclusion **23**

References **24**

A Statement on AI Contributions **25**

B Interactive Dashboard Usability Survey **26**

1 Introduction

In recent years, numerous teaching approaches have been developed to improve the effectiveness of computer science education for young learners. Among these, hands-on learning, based on actively writing and testing code, has proven to be particularly effective, as it encourages students to engage in problem-solving and decision-making processes. However, practical experience alone is not sufficient. Without a solid understanding of underlying concepts, students struggle to write functional and robust code. Therefore, an effective learning process requires a balanced integration of both theory and practice. The best way to learn complex concepts is to combine theory with immediate practical application, which helps learners to understand and remember what they are learning.

Educational platforms such as PyTamaro-Web aim to support this approach by enabling instructors to author custom curricula and programming activities. This environment allows users to immediately apply newly acquired theoretical concepts through short practical exercises [6]. This methodology not only facilitates learning but also generates a large amount of executable data that can potentially be analyzed to assess users' progress.

Despite the availability of such data, instructors currently face significant limitations. Because they author the assignments, one might expect that they could easily compare user submissions with reference solutions, evaluate exercise difficulty, and iteratively improve their curricula. In reality, the data stored within the PyTamaro database remain in a raw and unprocessed format, with no dedicated interface for analytical visualization. Consequently, valuable insights into user learning behaviors remain unknown. As a result, instructors are unable to effectively compare solutions, identify common errors, or detect learning difficulties, and therefore lack a systematic way to obtain feedback on user performance.

To bridge this gap between raw data and actionable insights, this thesis introduces the ***Interactive PyTamaro-Web Dashboard***. The proposed system is specifically designed to address this visibility problem by implementing an automated end-to-end data pipeline. The system extracts user execution histories, performs large-scale static analysis, and compiles the results into an interactive, offline-ready HTML dashboard. This tool enables instructors to explore their curricula and activities in detail, analyze user performance, examine common errors, and compare initial implementations with user submissions. Overall, it provides valuable insights into users' problem-solving processes and supports data-driven improvements in teaching practices.

1.1 Key Technical Contributions

The primary objective of this thesis is to transform raw educational telemetry into accessible, actionable insights. To achieve this, the project introduces the following key technical contributions:

- **Data Extraction and Fusion Pipeline:** Design and implement a backend architecture that integrates structured relational data (PostgreSQL) with structured file-system assets (Jupyter Notebooks/JSON), reconstructing the full context of user programming sessions.
- **Optimized Static Analysis Engine:** Development of an efficient batch-processing pipeline that uses the `ty` type checker [2]. A global execution cache eliminates redundant computations, significantly reducing analysis time for large datasets.
- **Offline Interactive Dashboard:** Creation of a fully static, responsive frontend built with Jinja2 and Tailwind CSS, integrating the Monaco Editor for syntax highlighting, code comparison, and error visualization without requiring a backend server.

1.2 Structure of the Thesis

The thesis is organized as follows. Chapter 2 introduces the PyTamaro-Web educational environment and its dual-storage architecture. It highlights the pedagogical blind spot instructors face and details the technical challenges related to extracting, analyzing, and reconstructing anonymous user telemetry. Chapter 3 describes the design and implementation of the proposed analytics pipeline. It details the backend data extraction process, the scalable static analysis engine utilizing global caching, and the generation of the static, offline-ready frontend dashboard. Chapter 4 evaluates the system through a validation study conducted with an instructor. It assesses the dashboard's pedagogical effectiveness, evaluates its usability, and identifies areas for future workflow improvements. Finally, Chapter 5 concludes the thesis by summarizing the main contributions and discussing directions for future work based on the evaluation feedback.

Additionally, the appendices provide supplementary material: Appendix A outlines the generative AI tools used during the project, and Appendix B contains the full interactive dashboard usability survey distributed for the evaluation.

2 Background and Problem Statement

In order to understand the technical challenges involved in generating interactive dashboard starting from a web educational platform, it is necessary to understand the environment in which the data is generated. This chapter aims to explain the PyTamaro-Web [6] platform, detailing its pedagogical purpose, its underlying architecture, and the specific problem this thesis aims to solve.

2.1 Related Work in Educational Telemetry

In order to understand how to process and visualize large-scale educational data, it is necessary to examine previous efforts in mining code generated by students.

A significant example is the work of Altadmri and Brown, who examined beginners' programming behavior by analyzing 37 million compilation events from over 250,000 users worldwide using the Blackbox dataset [1]. By tracking syntax, semantic and type errors over the academic year, they identified the most frequent mistakes and those that took the longest to fix. The study shows how large-scale telemetry can reveal significant trends in the learning process. For example, it demonstrates that semantic errors persist more than syntax errors as students progress through a curriculum. While their approach to reconstructing the evolution of source files between compilations is similar to the process of reconstructing chronological user sessions, their work consists of a large-scale statistical study. In contrast, this project aims to develop an interactive tool that provides instructors with actionable insights based on static analysis generated within PyTamaro-Web.

Another related approach is OverCode, a system designed to visualize thousands of programming solutions on a large scale [5]. OverCode groups similar solutions into "stacks" by combining static analysis with dynamic program tracing, clustering code according to how variables behave during execution. This approach allows instructors to explore different solution strategies and recognize common implementation patterns adopted by students.

Even though both OverCode and the dashboard proposed in this thesis aim to aggregate large amounts of educational data and provide them to instructors through a user-friendly interface, they differ in approach and purpose. One important difference is that OverCode visualizes only the final solutions that have been marked as correct by an autograder. The PyTamaro-Web dashboard, on the other hand, is designed to analyze the entire learning process, including intermediate attempts, repeated mistakes and the chronological evolution of code. Another distinction lies in the analysis methodology. OverCode uses dynamic analysis to trace variable behavior by executing test cases, while the approach proposed in this thesis uses static analysis performed during an offline generation phase. A further difference is that OverCode clusters final submissions from many users, while the PyTamaro-Web dashboard's core challenge is to reconstruct complete user sessions over time.

These studies provide valuable insights into the importance and usefulness of collecting and analyzing telemetry data in large-scale education. However, they also highlight a common issue with many modern learning platforms: the data is often inaccessible to instructors, who could use it to refine their teaching strategies. The dashboard proposed in this thesis aims to resolve this issue using the telemetry data generated by PyTamaro-Web to create visualizations that provide instructors with concrete insights.

2.2 The PyTamaro-Web Educational Environment

PyTamaro-Web is an interactive, web-based educational platform designed to teach programming concepts through active, hands-on learning. Instead of traditional textbooks or passive lecture materials, the platform enables users to write and execute code directly within their browsers, receiving immediate feedback after an execution. Instructors utilize the platform to supplement traditional computer science classes by designing personalized curricula and interactive activities for their students, as well as for users worldwide who are interested in using the platform. The primary educational structure on the site is centered around the *Curricula* page [3, Chapter 7].

A Curriculum is composed of individual Activities, which consist of a hybrid structure of descriptive text and executable code cells. The concept of a Curriculum is to group related activities in a progressive sequence. As users advance from one activity to the next, they construct pieces of code and foundational knowledge. This guided approach smoothly transitions the user from foundational concepts to complex final results. This progression is facilitated by the internal structure of the activities themselves. Instructors use text sections to explain the theoretical logic and necessary background knowledge, immediately followed by interactive code cells. To make the learning process easier, these cells are not usually left blank at the start. Instead, instructors fill them in beforehand with sample code or guided exercises. This allows users to apply the newly acquired theoretical concepts, either independently or under guidance, seconds after reading about them.

To accommodate different educational scenarios and users, an Activity can have multiple Variants. Variants allow instructors to update an activity, fix its difficulty, or translate it into different languages to improve accessibility. Every

variant provides a fully interactive, live execution environment where users can read the problem description, write Python code, and run it. The execution process is supported by a real-time static analysis engine that continuously inspects the code and provides immediate feedback on syntactic or semantic errors [3, Chapter 7.5]. To better support different learning approaches, users can choose among three analysis modes:

- **None**: No static errors are highlighted before code execution. In addition, syntax highlighting is disabled (see Figure 2.1).
- **Syntax**: Syntactic errors are highlighted before code execution (see Figure 2.2). Basic syntax highlighting is enabled to distinguish language tokens.
- **Syntax & Semantics**: Both syntactic and semantic errors are highlighted before execution. Additionally, the editor applies specific highlighting to distinguish tokens such as function declarations and variables (see Figure 2.3).

This immediate feedback, combined with the analysis modes, creates an environment in which users can test hypotheses and learn through trial and error [3, Chapter 8].

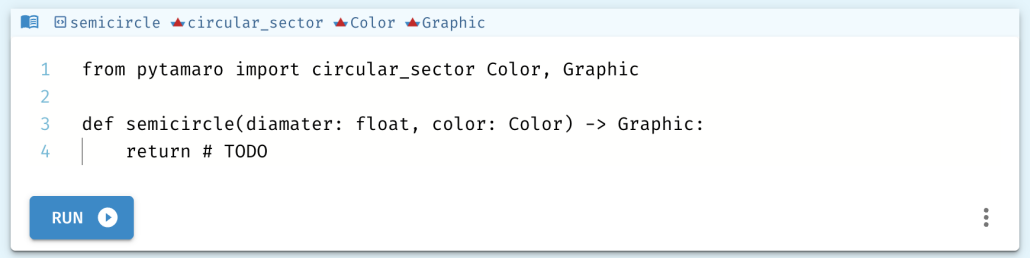


Figure 2.1. Example of the PyTamaro-Web editor interface with no active error feedback or syntax highlighting.

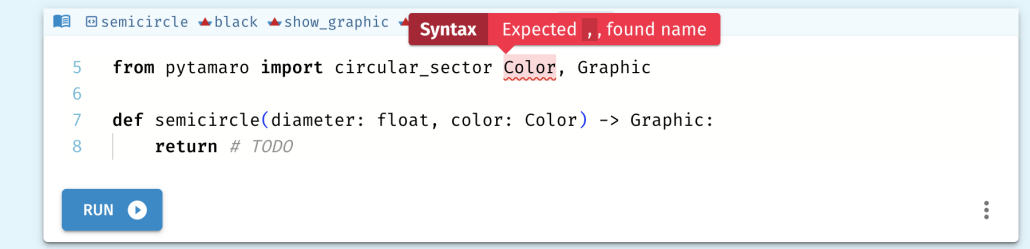


Figure 2.2. Example of syntax error detection in the PyTamaro-Web editor.

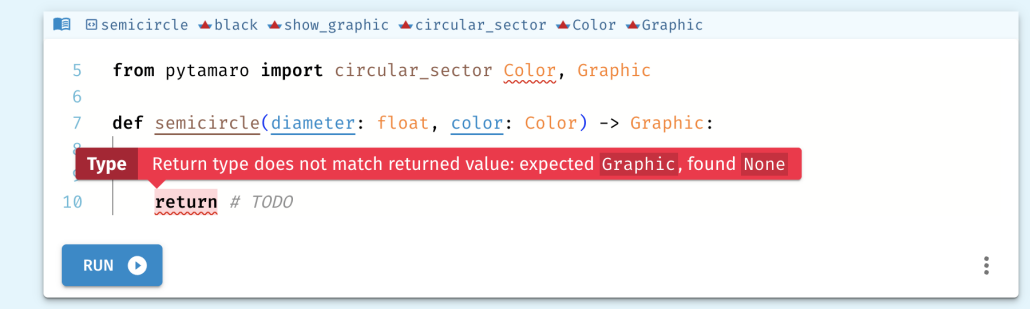


Figure 2.3. Example of combined syntax and semantic analysis feedback with highlighted code elements in the PyTamaro-Web editor.

2.3 Dual-Storage Platform Architecture

PyTamaro-Web operates on a dual-storage architecture, splitting the instructor's assignment from the users' interactive telemetry [3, Chapter 7]. As illustrated in Figure 2.4, the platform strictly enforces anonymous data collection [3, Section 7.5], there is no login or authentication requirement for users. While this ensures that no sensitive personal information is collected or stored, it introduces significant complexity regarding data relationships. Because activities

are generated on the file system, external to the database, and users operate entirely anonymously, retrieving and linking a cohesive timeline of a specific user's session is a highly complex process.

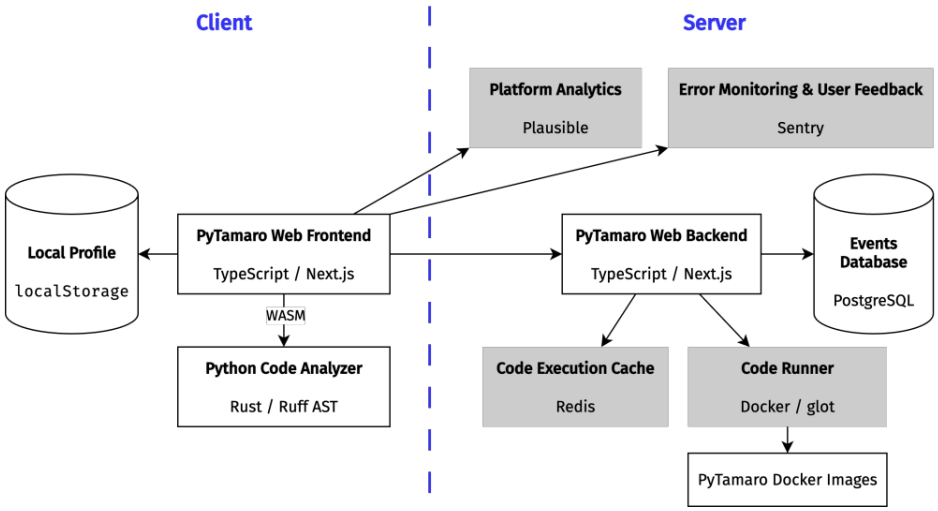


Figure 2.4. Architecture diagram of the PyTamaro-Web platform. The separation between the client-side local profile and the server-side events database illustrates the dual-storage nature of the system. Figure reproduced from [3, Section 7.5].

2.3.1 File System Content and Instructor Repositories

Every instructor collaborating on PyTamaro-Web must maintain a highly structured repository to store all curricula, activities, and supplementary files. The content displayed on the platform is determined by three primary directories within this repository. The *curricula* directory is populated by JSON files which define the structural hierarchy and grouping of the activities. The *activities* directory contains the actual Jupyter Notebooks (`.ipynb` files) [10] that define the exercises. Finally, the *extra Python files* directory contains shared functions or methods required by users for some specific activities.

These notebooks are authored by instructors and combine Markdown cells with executable Python cells. The Markdown cells provide instructions, theoretical explanations, and examples. On the other hand, the Python cells provide the interactive workspace, often containing initial base code and `TODO` comments where users write and execute their solutions. All of these assets are stored directly in the PyTamaro repository and are accessed via the local file system.

2.3.2 Relational Telemetry and Anonymous Event Logging

While the assignments live on the file system, user interactions within the platform are logged in a PostgreSQL database. This telemetry captures a wide range of actions, from basic navigation between activities in a curriculum to detailed code execution events, including whether specific toolbox functions or extra Python files were utilized during the run [4]. Because the platform operates under strict privacy rules and uses anonymous sessions, tracking what was done, by whom and in what context relies entirely on these relational event logs. This tracking mechanism is centered on three primary tables, as illustrated in the Entity-Relationship diagram in Figure 2.5:

- **Activities**: This table stores all metadata associated with a specific activity variant.
- **Events**: This table logs every interaction a user has with the platform. It serves as the primary key for reconstructing a user's session, allowing the system to retrieve the exact timestamp and context of a specific action.
- **Code Execution Sources**: This table stores the exact code that the user has executed. It also captures a snapshot of the initial cell code provided by the instructor at that moment. This enables the system to establish a reliable baseline for comparison, even if the instructor subsequently updates or alters the original file system activity.

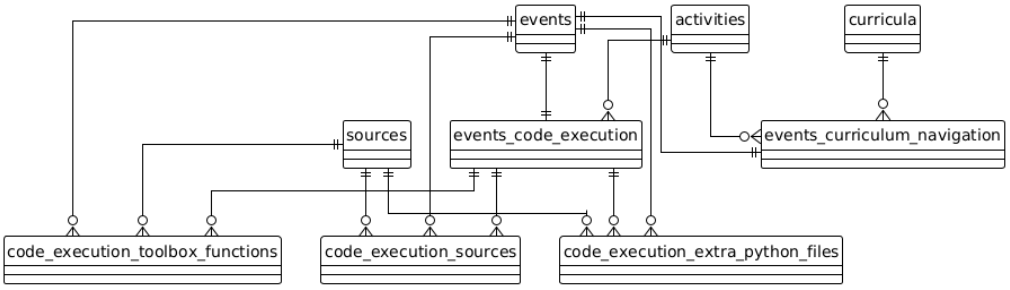


Figure 2.5. Entity-Relationship diagram illustrating the highly relational structure of the PyTamaro-Web telemetry database, highlighting the central role of the events table.

2.4 The Instructor's Blind Spot: Bridging the Feedback Gap

While PyTamaro-Web is very effective at capturing user telemetry, the absence of an automated pipeline to transform this raw execution data into accessible insights creates a critical functional gap. Every interaction, cell execution and code modification is stored in the database. However, these records remain there and cannot be read or viewed in a dedicated UI. Extracting meaningful information requires the manual creation of complex database queries and the cross-referencing of thousands of disconnected events. Additionally, every single user's session must be reconstructed manually.

Since PyTamaro-Web has been designed for use in lessons, the inability to visualize generated data represents a significant blind spot in teaching. Instructors are not able to get a clear picture of their students' understanding. To provide an effective learning environment, teachers need to be able to assess the difficulty of a topic, rather than just relying on intuition. They need a full picture of the learning process: how a student's code evolves over time, which syntactic or semantic errors cause recurring problems, and how final submissions differ from the initial code. Access to these detailed metrics is essential for determining whether the curriculum is structured appropriately, and for identifying which activities require revision, updating, or complete redesign to improve student comprehension.

Since the telemetry remains in raw format and there is no dedicated interface for analytical visualization, the platform essentially acts as a *black box* from the instructor's perspective. They don't have a systematic way of getting feedback on how the class is performing. This makes it difficult to identify weaknesses in current teaching strategies and prevents the iterative improvement of curricula over time. Providing instructors with an intuitive, interactive, visual representation of student progress would be highly beneficial, but it would introduce a series of complex technical challenges regarding data aggregation, static analysis, and scalable web architecture.

2.5 Technical Challenges

The pedagogical limitations described in the previous sections, including the instructor's inability to track student progress, identify recurring errors and improve curricula iteratively, are not caused only by the absence of visualizations. They emerge from a broader set of technical challenges related to educational telemetry analysis. Transforming raw execution data into meaningful insights requires reconstructing fragmented user interactions, processing large volumes of source code efficiently, and presenting the resulting information through an accessible interface. The development of the Interactive PyTamaro-Web Dashboard therefore required addressing challenges relating to data reconstruction, scalable static analysis, and interactive offline visualization.

2.5.1 Reconstructing User Sessions

One of the challenges of the project was reconstructing coherent user sessions from fragmented telemetry data. PyTamaro-Web records a large number of user interactions. These events are distributed across multiple independent data sources and are not directly organized into complete execution timelines. This complexity is also increased by the platform's anonymous architecture. Since users are not authenticated, the system cannot rely on traditional user profiles or persistent identifiers to reconstruct activity histories. Instead, the relationships between executions, activities, variants, and auxiliary resources must be inferred from the telemetry itself.

In addition, the platform separates instructional content from execution data. While user interactions are stored in the PostgreSQL database, the original assignments, notebook structures, and supplementary files remain stored in the instructor's repository on the local file system. As a consequence, generating meaningful analytics requires combining relational telemetry with external notebook assets to recreate the original context in which each execution occurred. Without this reconstruction process, the raw telemetry would provide only isolated execution events rather than a meaningful representation of user progress and learning behavior.

Understanding user behavior requires more than simply retrieving the final submitted code. It is necessary to reconstruct the chronological evolution of each execution session, including intermediate attempts, repeated mistakes and progressive changes to the code. Without this contextual information, the telemetry would only show isolated execution events, rather than providing a meaningful representation of the user's learning process.

Therefore, the challenge was not just collecting data, but also transforming disconnected educational telemetry into a structured representation that can describe how users interact with programming activities over time.

2.5.2 Scalability of Static Code Analysis

Another challenge was the scalability of static analysis. PyTamaro-Web supports real-time syntax and semantic analysis during code execution, meaning that static analysis data becomes an important part of the learning process. However, generating analytical insights for instructors requires processing not only users' final submissions, but also the complete history of their intermediate executions. Since a single activity may contain thousands of executions distributed across many users and variants, performing static analysis at scale introduces significant additional computational requirements.

This challenge is made more significant by the offline nature of the generated platform. Instead of performing analysis dynamically through backend requests, as it is possible with a traditional web application, the proposed system must perform all computations during the generation phase. Therefore, the pipeline must efficiently process large datasets while avoiding excessive execution times and redundant computations. Designing a scalable analysis workflow has therefore become a central requirement of the project.

2.5.3 Constraints of an Offline Interactive Dashboard

Finally, the generation and distribution of the resulting analysis was also a significant challenge. To maximize accessibility for instructors, the dashboard was designed as a static website that could be shared and opened locally, eliminating the need for server deployment or backend infrastructure. While this approach simplifies portability and distribution, it also introduces significant limitations to the frontend. Features that are usually handled by backend services, such as filtering data, navigating execution histories or dynamically comparing code submissions, must instead be implemented entirely within the browser. At the same time, the dashboard had to present large volumes of educational telemetry while remaining responsive and easy to understand. Careful design of the interaction model and organization of the visual components was required to ensure that the generated dashboard remained both comprehensive and user-friendly.

3 Designing the Analytics Pipeline and Dashboard

The challenges outlined in Chapter 2 guided the design of the proposed analytics pipeline. In particular, the system architecture was designed to address three primary requirements: efficient reconstruction of educational telemetry, scalable execution analysis, and generation of a portable interactive dashboard.

3.1 High-Level System Architecture

Developing a solid pipeline required a clear system architecture. Initially, the iterations were based on a simple model, but as the complexity of the data extraction and transformation process increased, they required a well-structured approach. The final architecture was designed to sequentially process data extraction, static analysis, and HTML pages generation, ensuring that all retrieved information is efficiently transformed into an interactive dashboard. Figure 3.1 illustrates the high-level execution flow of the system.

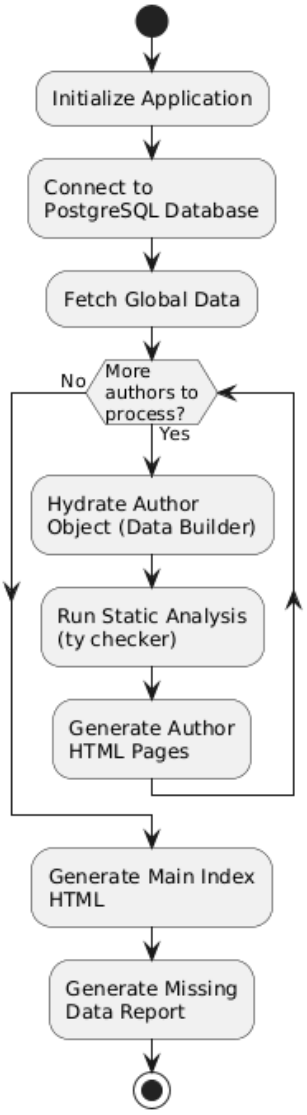


Figure 3.1. High-level execution of the pipeline for generating the Interactive PyTamaro-Web Dashboard.

3.2 CLI-Based Application Initialization

The main pipeline is initialized via a command-line interface (CLI). To execute it, the environment requires access to a local instance of the PyTamaro-Web database. The application is triggered using the command `python3 -m src.main`. It is implemented using the `Click` library [8], which provides a declarative decorator-based approach to CLI creation, significantly reducing manual argument parsing and validation logic.

The application requires the `PATH` to the local `pytamaro-web` data folder as a mandatory positional argument. The CLI supports the following configuration options:

- `-a` or `--authors`: A comma-separated list of authors to process. Within the context of the system's architecture and CLI, the term "author" is used to designate the instructor who created the educational activities and curricula. If specified, the system generates insights exclusively for those individuals, otherwise, the system defaults to processing all authors with existing activity or curriculum records in the database.
- `-d` or `--debug`: Enables debug-level logging, providing detailed execution traces and internal status updates beyond standard informational logs.

The general execution schema is defined as:

```
python3 -m src.main [OPTIONS] PATH
```

A concrete example enabling debug mode for specific authors is shown below:

```
python3 -m src.main -a author_1,author_2 -d ../../data/
```

3.3 Database-Related Optimizations

Upon initialization, the system establishes a persistent connection to the PostgreSQL database to support data retrieval throughout the entire execution pipeline. This operation is managed by the `DatabaseManager` class, implemented in the `database.py` module, which encapsulates all database interactions and query execution logic.

The PyTamaro-Web telemetry model is highly relational, centered around the continuous recording of user interactions (as illustrated previously in Figure 2.5). Most entities are connected via the `events` table, which acts as the main entry point for reconstructing execution histories and activity timelines. Consequently, generating meaningful analytics requires the combination of information distributed across multiple tables.

In order to reconstruct a complete execution, the extraction layer has to extract not only the executed user code, but also the related metadata associated with that execution. This includes activity variants, timestamps, execution sources, toolbox dependencies and any supplementary Python sources required for the original activity [4]. Since these relationships are spread across the database, this process needs to be able to rebuild the original session using structured relational queries.

During the early development phase, the prototype repeatedly opened and closed database connections for each individual fetch operation. Even though it was correct, it introduced overhead and reduced performance during the extraction workflows. To resolve this issue, the architecture was refactored so that a single persistent database connection would be maintained for the entire pipeline. This modification improved stability and minimized the risk of failure during data retrieval.

Another significant challenge was the amount of telemetry data processed by the system. Loading very large result sets into the memory can quickly become inefficient and unstable. To resolve this, the extraction layer uses a chunking strategy, splitting large queries into smaller subsets. Furthermore, to handle the complexity of the queries themselves, Common Table Expressions (CTEs) [13] were used. These allow the database to filter, aggregate and reorganize the temporary intermediate results before the final output is constructed. This approach reduces the complexity of some operations inside the Python application.

In addition, the system uses native PostgreSQL functions, such as `json_build_object` [12], to organize data directly within the database. Instead of manually converting raw SQL rows into nested Python dictionaries, the database generates structured JSON objects ready for use by the pipeline. This optimization reduces the overhead of intermediate transformations and simplifies the logic in the data modeling phase. This maintains efficiency in terms of performance and memory throughout the entire pipeline.

3.4 Object-Oriented Data Modeling

Once the data has been extracted from the database, the next step is to transform the raw information into a structured, object-oriented representation. This layer is implemented using *Python Data Classes*, which offer a type-safe way of modeling entities. This hierarchy plays a key role, it is designed to mirror the logical structure of the frontend. Since the dashboard pages are generated using Jinja2 templates, maintaining a coherent structure simplifies the rendering logic. Figure 3.2 illustrates the high-level class architecture.

To avoid having to deal with extremely complex database joins during the generation phase, the data is organized into the following hierarchical models:

`Author`: The root node of the hierarchy. It encapsulates a specific instructor's profile and acts as the entry point for their associated curricula and activities.

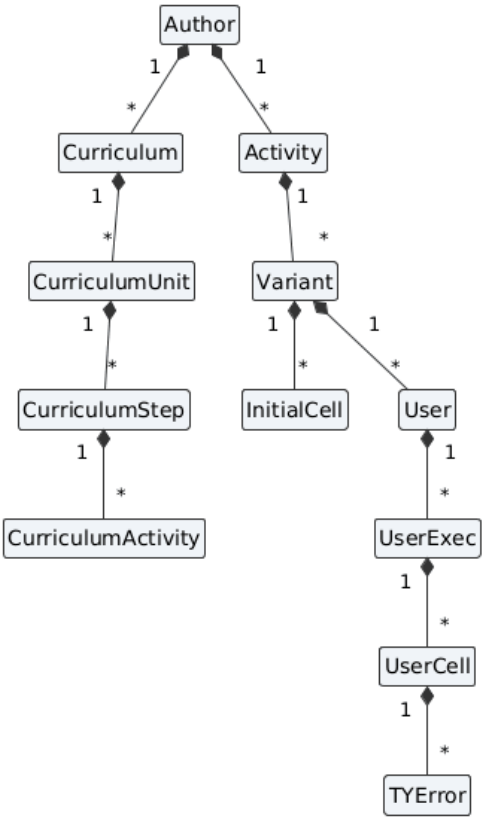


Figure 3.2. UML Class Diagram representation of the Pipeline Objects hierarchy.

`Curriculum`: Represents a collection of activities. It contains all the information about the curriculum, used as a container for multiple `CurriculumUnit` objects.

`CurriculumUnit`: Represents a specific chapter or thematic block within a curriculum. It stores a collection of `CurriculumStep` objects that organize the learning flow.

`CurriculumStep`: Acts as a structural wrapper within a unit, containing a list of specific `CurriculumActivity` items that the Variant must complete in sequence.

`CurriculumActivity`: A concise representation of an activity, specifically designed for the curriculum dashboard. Rather than loading full activity data, it stores basic metrics and an `activity_url` that links the user to the detailed activity page.

`Activity`: Represents a specific programming assignment or task assigned to users.

`Variant`: A specific type of activity, since each of them may have multiple versions and be written in many languages (English, German, Italian and French). It serves as a container for both the initial code provided by the instructor and all the attempts made by the users.

`InitialCell`: The original, unmodified code blocks provided by the instructor. This serves as the baseline for calculating code diffs in the frontend.

`User`: Represents an individual user participating in the activity variant, containing their complete chronological history of code executions.

`UserExec`: A snapshot of a single execution by the user. It stores the timestamp, the analysis mode selected by the user and the specific code cells submitted at that moment.

`UserCell`: The actual block of code executed by the user with its initial code to be more accurate between the new version shown on frontend and the one saw by the user at that time.

`TYError`: If a `UserCell` contains syntax or type errors during static analysis, this object captures the specific error message, line number, and column. This allows the frontend to visually display a label where it describes the specific error.

3.5 Data Builder and Object Hydration

After defining the object hierarchy, the next phase is populating the objects. This process is managed by the `DataBuilder` class, which is implemented in the `builder.py` module. The `DataBuilder` class is responsible for transforming query outputs into Python objects that are ready for frontend rendering. The construction of analytical objects requires the combination of two workflows: activity instantiation and curriculum aggregation.

3.5.1 Activity and Variant Reconstruction

The hydration process begins with `Activity` and `Variant` reconstruction. For each activity, the pipeline first identifies those users who have executed the corresponding activity at least once. Then, the "builder" reconstructs the chronological execution history associated with each variant. While the database stores the chronological history of each user's executions, the instructor's initial code is stored externally into `pytamaro-web` repository as Jupyter Notebook files. Therefore, reconstructing a complete context requires integrating also the file-system assets.

Using the custom `extract_code_cells` utility function, the "builder" navigates the repository to find the source notebooks (`.ipynb` files). Once identified, the notebook is parsed directly from its JSON structure. During this phase, it filters out all Markdown cells and drops any code cells explicitly tagged as `#teacher-only`. This tag is a custom convention used within the platform's authoring workflow. Instructors typically write the complete, functional code to verify the final result before removing specific parts to create the user's exercise. The `#teacher-only` tag ensures that these solutions remain hidden from the final rendered output. Concurrently, the utility accesses adjacent configuration files to extract accurate metadata, such as the activity's real display name. The "builder" can only launch the static analysis pipeline (detailed in Section 3.6) after merging the baseline code with the complete execution history.

Because the repository evolves, there may be inconsistencies between the two sources. In particular, activities referenced in the database may no longer exist locally because they have been renamed or deleted, or are missing from the downloaded repository version. To handle these missing assets, the system uses a global variable, `_MISSING_DATA_FOLDER`, to track and save them. Later on in the process, the pipeline uses this variable to generate a report called `missing_data_folder.txt`, which lists all activities that don't have a physical folder. During the early stages of development, this mechanism was useful for validating the behavior of the function and identifying any discrepancies in spelling between the database activity slugs and the actual repository folder names.

3.5.2 Curriculum Aggregation and Metric Extraction

Once the `Activity` and `Variant` structures have been successfully reconstructed, the "builder" proceeds with the `Curriculum` aggregation. This involves parsing the original configuration files stored in the repository and linking them to the just-instantiated activity objects.

This process preserves the organization defined by instructors, introducing additional analytical metrics computed during the pipeline execution. For each activity in a curriculum, the pipeline extracts three core metrics: the total number of user executions, direct links to detailed activity page, and the three most frequent static errors (calculated based on aggregated error statistics for the variant). This instantly highlights the most challenging parts of a curriculum for instructors.

3.6 Optimized Static Analysis and Global Caching

The execution of large-scale static analysis on user submissions was one of the most computationally challenging aspects of the project. PyTamaro-Web integrates the Python type checker `ty` to provide syntactic and semantic feedback during programming activities [2] [3, Section 7.5]. Extending this analysis to tens of thousands of historical user executions introduced significant scalability challenges within the pipeline.
To provide context on the database's size, the final exported database included 13 authors, 51 curricula, and 499 unique activities. A total of 106,080 user executions were processed across these activities, generating 156,625 static analysis diagnostics. In order to achieve acceptable performance over this volume of data, the integration of the `ty` checker underwent several architectural iterations:

1. **Per-Execution Analysis:** In the initial implementation, the static analyzer was invoked for each user execution via a dedicated `subprocess` call. However, because spawning a new `subprocess` is expensive, this introduced performance problems for some activity variants that took more than ten minutes to process.

2. **Per-Variant Batching:** Instead of analyzing submissions individually, the pipeline grouped all execution files belonging to the same activity variant into temporary directories and invoked the `ty` checker only once per variant. This optimization reduced the number of `subprocess` initializations and improved processing speed,

only a few minutes are now required per variant. However, many user submissions were identical because they reused template code or produced analogous incorrect solutions when practicing the given exercises. Therefore, repeatedly analyzing equivalent submissions resulted in unnecessary computations.

3. **Global Caching and Batch Execution:** To avoid this redundancy, the final architecture introduced a global execution cache that can identify execution patterns that have already been analyzed. For each execution sequence, the system generates a unique signature based on a combination of the following:

   - the user code,
   - the toolbox source [4],
   - and the supplementary Python dependencies (`extra_python_files`) associated with the activity variant.

   If an equivalent execution has already been processed, the pipeline reuses the cached analysis results without invoking the static analyzer again. Out of the total of 106,080 user executions in the dataset, the caching mechanism successfully reduced the workload to just 76,985 unique execution sequences, avoiding almost 30,000 redundant computations. Only executions not seen before are included in later analysis runs. To maximize efficiency where possible, all unique, unprocessed executions are collected in a single, temporary analysis directory. The `ty` checker is then invoked once for the entire activity, rather than for each variant.

Once the analysis is complete, the pipeline parses the generated JSON output, normalizes internal module names (e.g., reformatting temporary `toolbox_X` aliases back to the default `toolbox` [4]) and maps errors back to the specific cell index. This enables correct frontend visualization during the rendering process.

The final optimization pipeline has drastically reduced execution times. Workloads that initially took several minutes per variant to complete can now process the entire database export (over 100,000 executions) in approximately 76 seconds. Additionally, these optimizations enabled large-scale offline analysis and ensured consistency between the results of the static analysis and users' execution histories.

3.7 Static Frontend Architecture and Visualizations

Once the data extraction and modeling phases are complete, the final step is to present the information in an intuitive, interactive way through a UI. Because the analytical data is pre-computed offline, a live backend server is unnecessary. While modern frontend frameworks such as `React` or `Vue.js` offer clear benefits for client-side development, such as component reusability and robust state management, integrating their specific build environments (e.g. Node.js package managers) into a Python-only analytical pipeline would create significant unnecessary complexity. Instead, a static site generation approach was adopted using `Jinja2` [9], a template engine for Python. This makes it possible to create a highly portable dashboard using only `HTML`, `CSS`, and `JavaScript` files, allowing any user to open and explore the reports directly in a web browser without requiring a local server or additional software. The rendering pipeline is managed by the `generate_report.py` module. Before the workflow can begin, the environment must be set up to correctly store the generated output files. By invoking the `setup_report_assets` function, the system cleans up any previous builds, copies all the necessary JavaScript assets, and finally executes a `subprocess` to compile the `Tailwind CSS` stylesheet.

For visual design, `Tailwind CSS` [11] was selected because it is a very modern styling framework that doesn't require any additional packages or dependencies to be run, it can be set up using a straightforward command-line execution. It enables the creation of a sophisticated user interface by applying pre-defined class names instead of generating CSS code manually, which reduces the time required to write and maintain manual CSS rules. In addition, it enables the management of the responsiveness of the UI thanks to built-in media query classes. It also provides automatic color theme management. By appending a `dark:` prefix to assigned classes, the interface detects the author's browser theme preference and switches between light and dark modes. Although it is executed by command-line, it does not compromise the performance of the entire pipeline. The Tailwind compiler scans the templates to generate an optimized stylesheet containing only the used classes.

As illustrated in Figure 3.3, the frontend architecture of the dashboard is composed of four template pages. The `activity_page.html` template is the most complex and crucial component of the system. While `index.html` (see Section 3.7.1) provides an overview of all processed authors and works as the global entry point, the remaining three templates have specific, well-defined roles in the pipeline. The "Author Dashboard" page is a main page where all the activities and curricula for a specific author are listed (Section 3.7.2). From this page, the user can navigate deeper into the dashboard's subsections via the "Curriculum" or "Activity" pages, which provide detailed reporting metrics for a single curriculum (Section 3.7.3) or a specific activity (Section 3.7.4), respectively.

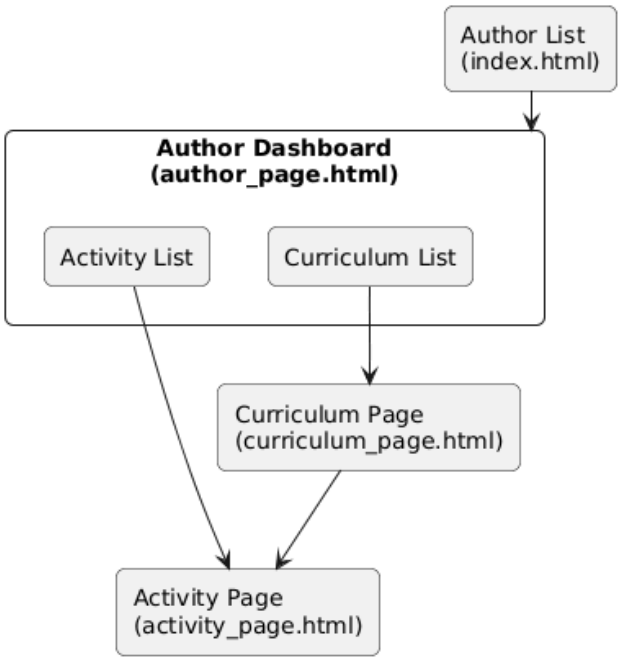


Figure 3.3. Navigation flow diagram illustrating the hierarchical structure and routing between the generated static HTML pages.

3.7.1 Author List

The `index.html` page, illustrated in Figure 3.4, works as the main entry point to the dashboard. It includes a summary table that shows all the authors that were processed during the pipeline execution, as well as some metrics, such as the total number of authors, activities, and curricula. The table displays the total number of activities and curricula for each individual author, the timestamps for their most recent curriculum interaction and user execution. If no historical data exists for a given field, the interface explicitly displays "*No data available*".

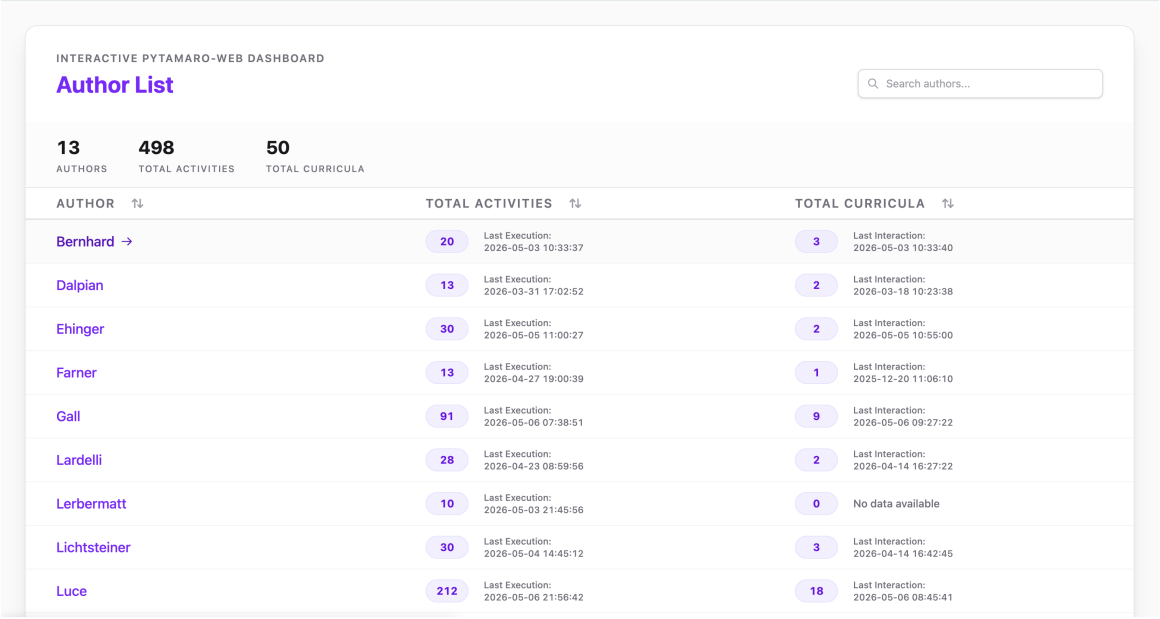


Figure 3.4. Screenshot of the Global Dashboard (`index.html`), which provides a high-level summary of all processed authors and displays the total number of activities and curricula generated.

In order to support future growth of the telemetry, we implemented an interactive sorting logic. In this way the authors can dynamically sort the table either alphabetically by author name or numerically by activity and curriculum counts. The sorting order can be toggled between ascending and descending by interacting with the column headers. The default setting is ascending alphabetical order.

Furthermore, a search bar was integrated to allow the author to filter the list by name. To optimize client-side

performance, the search input uses a debouncing strategy. It delays the execution of the JavaScript filtering function until the user stops typing, to reduce prevent excessive DOM updates. When an author selects an author's name, they are redirected to the author's dashboard page.

3.7.2 Author Dashboard: Activities and Curricula

The `author_page.html` template works as a central navigational hub for a specific author. The interface is divided into two distinct views: "Activity List" and "Curriculum List", both of which are organized into responsive grid layouts containing summary cards. This page also includes a debounced search bar that dynamically updates its filtering logic based on the currently active view. This guarantees that the search queries only target the visible dataset.

Activities List View

In the "Activities List" view (see Figure 3.5), the interface generates a dynamic card for each activity processed before by the pipeline. These cards display performance metrics. In particular, they show the total number of unique users who have attempted the activity, in relation to the author's total active user count. Additionally, the cards display the ***Average Error Rate***, a calculated ratio reflecting the mean number of static errors encountered per run.

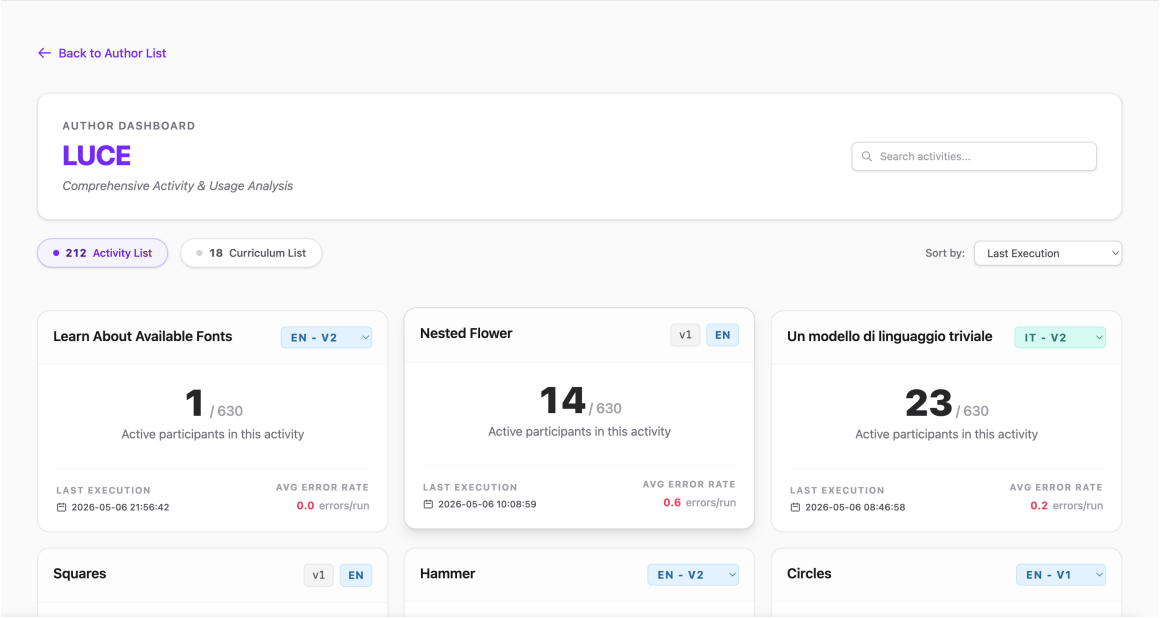


Figure 3.5. Activities List view.

To avoid duplication, the UI combines activity variants into a single card, then if an activity has multiple variants, the author can select a specific version using a dropdown menu. This triggers a JavaScript event that updates the card's statistics dynamically to reflect the data for that variant. This feature was implemented because previous versions required multiple cards for each variant of the same activity. Grouping the activity in this way is logical and prevents UI overload.

Finally, authors can sort these activities based on three criteria using a fixed-order sorting mechanism:

- ***Last Execution***: Sorts activities based on the most recent interaction.
- ***Most Users***: Sorts activities based on the highest user adoption.
- ***Highest Error Rate***: Sorts activities based on the highest average error rate.

All of these sorting algorithms organize the data in descending order. For any activity without an execution history, the interface displays a "No data available, never executed" message instead of null metrics (see Figure 3.6).

Curricula List View

In contrast, the "Curricula List" view (see Figure 3.7), provides a higher-level overview because raw telemetry regarding curriculum navigation offers less insight to the author. It displays the title of the curriculum, a truncated preview of its description, and the timestamp of the last user interaction. By clicking on a specific card, the author is redirected to the dedicated Curriculum Page, where in-depth analytical data is presented.

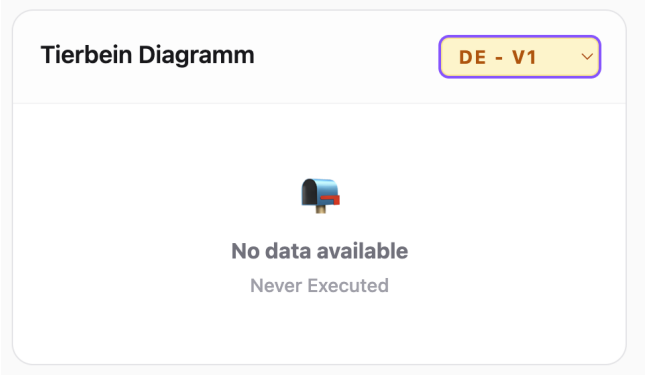


Figure 3.6. UI component displaying a "No data available, never executed" message for an activity without an execution history.

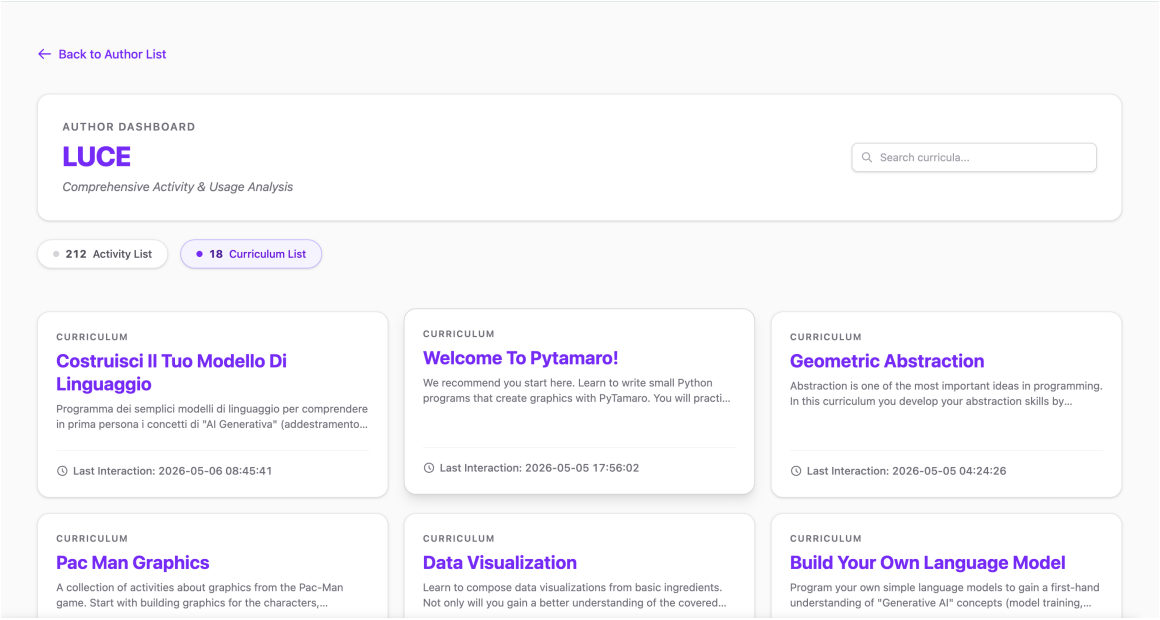


Figure 3.7. Curricula List view.

3.7.3 Curriculum Page

The `curriculum_page.html` template, illustrated in Figure 3.8, mirrors the main structure of the curriculum page on the PyTamaro-Web platform in order to preserve consistency. The page header displays the curriculum's title and full description. Long descriptions are truncated by default to optimize screen space. These can be expanded by toggling the "Show More" button. The curriculum is organized into expandable unit sections beneath the header. Each unit heading includes its title and a badge indicating the total number of activities within it. Expanding a unit displays the activities in the sequence defined by the author.

These steps consist of individual activity represented as interactive cards. These cards are designed to display key information, helping authors identify which activities may require immediate attention. Each card is divided into two sections: a header containing the activity's title, language and version, and a body displaying one of three possible interface states:

- ***No Static Errors Recorded:*** If the static analyzer found zero syntax or semantic errors across all runs, but the user execution telemetry exists, the UI displays a message indicating a clean execution record.
- ***No Activity Page:*** If the offline extraction pipeline did not find any data about the activity, the card indicates that no detailed report could be generated.
- ***Frequent Errors:*** Since errors are a normal part of the learning process, this is the most common state in most activities. In this state, the UI displays the three most frequent static errors detected during the analysis phase.

If they have been processed by the analyzer, all the cards work as direct links to their respective "Activity Page". This improves the dashboard's user-friendliness, enabling authors to access relevant activities in the curriculum without having to return to the "Author Dashboard", switch to the "Activity List" view, and perform a manual search.

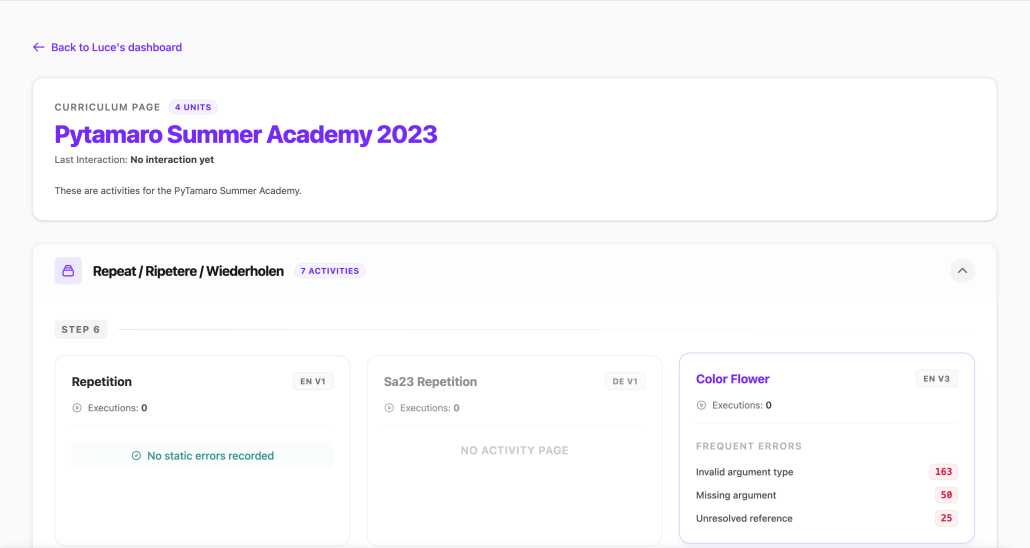


Figure 3.8. Screenshot of the Curriculum Page (`curriculum_page.html`), displaying a specific curriculum unit and its detailed activity cards.

3.7.4 Activity Page

As shown in Figure 3.9, the `activity_page.html` template is the analytical core of the platform. It allows the author to examine user execution histories in detail. This allows the author to check both individual user performance and the overall educational effectiveness of the exercise. The interface is divided into three sections: the header, the error analysis dashboard and the interactive execution viewer.

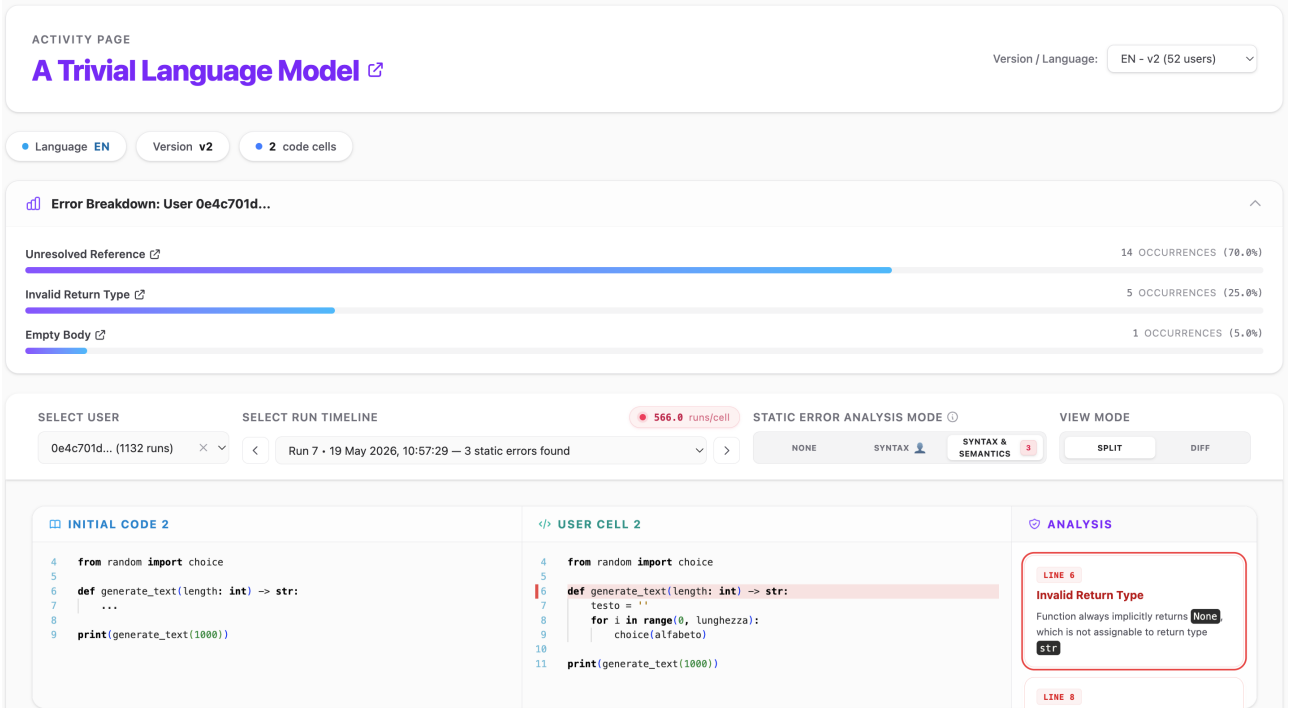


Figure 3.9. Screenshot of the Activity Details Page (`activity_page.html`), showing a detailed visualization of a specific activity.

Activity Header

The header provides an overview of the activity's data, displaying its title, language, version and the number of code cells provided by the author. If the activity has multiple versions or translations, the *Version/Language* dropdown menu allows the author to switch between variants. This instantly repopulates the page's data without requiring a reload, thanks to a JavaScript event. To help authors easily recall the original exercise instructions, the activity title works as a hyperlink that redirects to the PyTamaro-Web platform's activity page.

Error Analysis List

Below the header is located a collapsible error analysis list. This dynamic component operates in two primary states, depending on how the author interacts with it:

- ***Error Distribution***: By default, this state provides a general overview of the selected variant. This shows the most common static errors found among the whole user base, along with their number of occurrences and relative percentage.
- ***Error Breakdown per User***: When an author selects a user in the "Execution Viewer" below, the list switches to this state (as shown in Figure 3.9). It then recalculates and displays the error distribution specific to that user's execution history.

Due to the fact that the error classifications displayed on this list are generated directly by the `ty` static analyzer, the terminology can be very technical or ambiguous. To improve usability, every error name functions as a hyperlink to external documentation. Clicking on an error redirects the author to the official `ty` reference documentation, where they can access detailed explanations, root causes and code examples for that specific syntax or semantic error. Furthermore, the dashboard is able to handle scenarios where neither the variant nor the selected user has encountered any static errors. Instead of rendering an empty list, it provides messages such as "No errors were recorded for this variant" or "No errors were recorded for this user", ensuring that the author is aware that the run was completed without errors.

The interactive execution viewer, the most sophisticated component of the entire dashboard, is located in the lower section of the page. Designed to optimize chronological navigation through user runs, this viewer can isolate code errors and immediately highlight differences between the author's initial template and the user's submitted code.

Viewer Header Controls

The header of the execution viewer (see Figure 3.10) contains all the interactive data controllers. From left to right, the interface provides the following utilities:

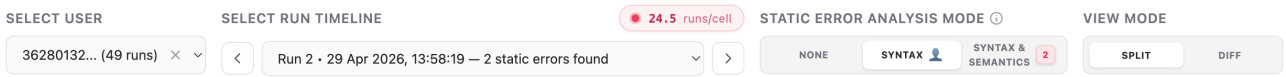
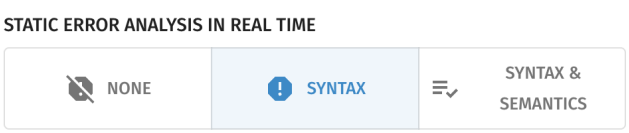
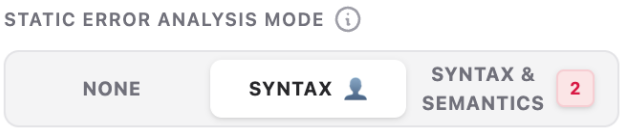


Figure 3.10. The Execution Viewer header features the User Selection drop-down menu, the Execution Timeline Navigator, the Static Analysis Mode toggle and the View Mode toggle.

- ***Select User***: This dropdown menu enables the author to select a specific user. To preserve anonymity while maintaining identification, a truncated UUID identifies users. The list is ordered in descending order by total number of runs.
- ***Select Run Timeline***: This control allows the author to view their users' execution history in chronological order. Left and right directional arrows offer rapid navigation through the history of runs, eliminating the need for manual dropdown interaction. Each run is indexed and provides an exact timestamp of the execution and the total number of static errors detected. Additionally, a dynamic "runs/cell" label displays the ratio of runs per code cell for the specific selected user. By default, the viewer is set to the user's most recent execution.
- ***Static Error Analysis Mode***: As detailed in Section 2.2, the original PyTamaro-Web platform allows users to customize their active analysis checker. This dashboard mirrors that functionality, providing three filtering states: "None", "Syntax", and "Syntax & Semantics". A dedicated icon indicates which mode the user selected at the time of execution. Toggling these buttons dynamically filters the "Analysis Column" to display only the relevant subset of errors. The UI for this component has been designed to maintain visual consistency with the PyTamaro-Web (see Figure 3.11).
- ***View Mode***: This control switches between the two display formats for the code editor. In "Split" mode, the initial code and the user's code are displayed in two independent columns. In contrast, the "Diff" mode combines these columns into a single view that highlights exact character and line modifications, which works as GitHub. This view significantly improves the author's ability to track user changes (see Figure 3.12).



(a) PyTamaro UI for selecting the static error analysis mode.

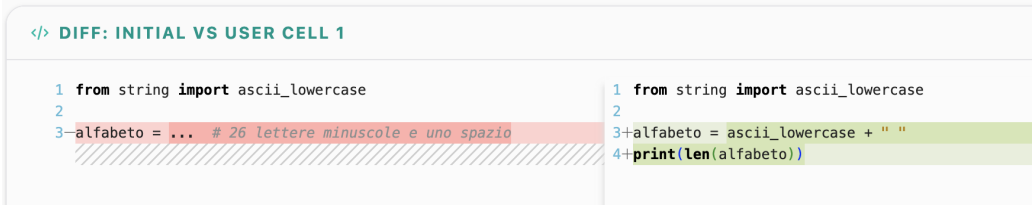


(b) Dashboard UI for selecting the static error analysis mode.

Figure 3.11. This comparison illustrates the design consistency between the original PyTamaro-Web platform (a) and the new dashboard interface (b) by showing the static error analysis mode UI components.



(a) Split mode view, the “Initial Code” and the “User Cell”.



(b) Diff mode view, initial and user code into a single column.

Figure 3.12. Comparison of the code editor view modes: the split mode (a) and the diff mode (b).

Code Viewer and Analysis Section

The central section of the execution viewer consists of the code visualization columns and the analysis column. The advanced code rendering features are powered by the Monaco Editor [7], which is the same core engine that powers Visual Studio Code.

In Split mode, the layout consists of two distinct columns. The “Initial Code” column remains static across all execution timelines for a given activity and represents the template provided to the user. The “User Cell” column dynamically updates to reflect the exact state of the code at the selected timestamp. If a user does not execute a specific cell during a given run, the UI renders a dedicated “Not Executed” placeholder to explicitly communicate this missing state to the author. To guarantee transparency, a warning tooltip informs the author that this template of the “Initial Code” was obtained from the current repository and may differ from the version originally seen by the user (see Figure 3.13).

On the other hand, the Diff mode merges these two states into a single “Diff: Initial vs User Cell” column. Using the Monaco Editor’s built-in diffing API, this view accurately shows the differences between the template and the submission.

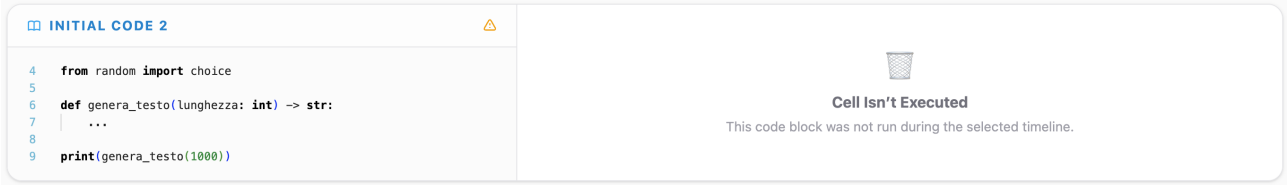


Figure 3.13. The UI message displayed when a cell was not executed. A warning tooltip is visible in the “Initial Code” column regarding version consistency.

Finally, the “Analysis” column displays interactive error cards that detailing the error type, the specific line number and a description (see Figure 3.14). When the author clicks on an error card, a JavaScript event enables the Monaco Editor to highlight the corresponding line of code in red, reducing the time required to locate it. This highlighting function is available in both *Split* and *Diff* view modes. Because the visibility of this column is linked to the “Analysis Mode” toggle, it uses dynamic empty states to show the author the correct status message:

- If **“None”** is active, the column hides all error cards and displays the message *“Analysis Mode is set to None”*.

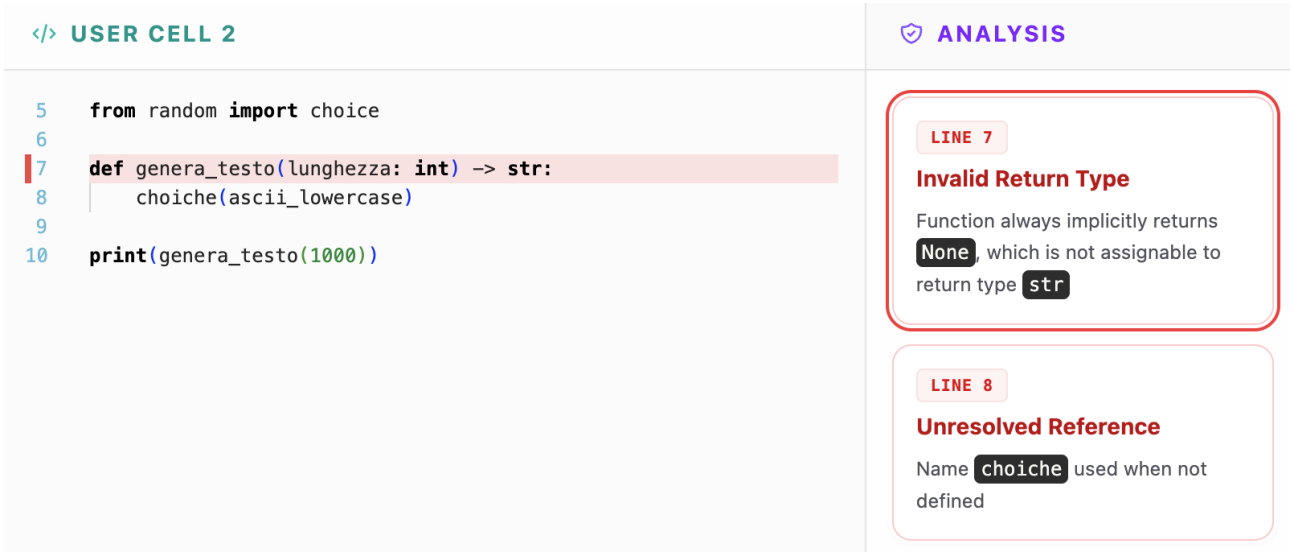
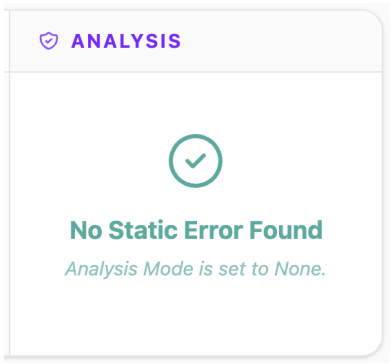


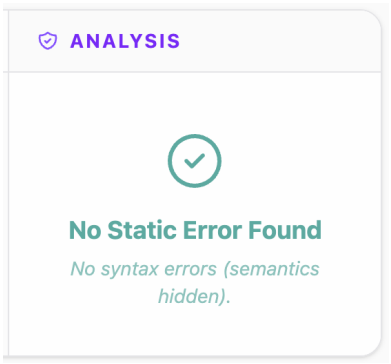
Figure 3.14. The analysis column displaying interactive error cards, including the line number, error type, and a description.

This clarifies why the UI is empty even if the underlying `ty` checker detected errors (see Figure 3.15a).

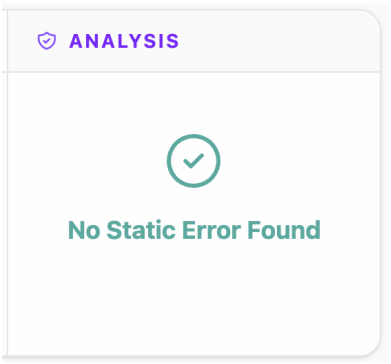
- If **"Syntax"** is active and the code is error-free, the column may display *"No Static Error Found"* (see Figure 3.15c). If there are semantic errors but no syntactic errors, it displays *"No syntax errors (semantics hidden)"*, which prevents the author from being confused (see Figure 3.15b). If there are syntactic errors, the standard error cards are displayed (see Figure 3.14).
- If **"Syntax & Semantics"** is active, the column will either render the relevant error cards or display the standard *"No Static Error Found"* state if the execution has no errors.



(a) Message when the analysis mode is set to "None".



(b) Message when no syntax errors, but semantics hidden.



(c) Message when no static errors are found in the cell.

Figure 3.15. Different states of the analysis column based on the selected analysis mode: "None" selected (a), a cell with no errors (b), and "Syntax" selected with semantic errors hidden (c).

4 System Evaluation

In collaboration with the authors, an evaluation study was conducted to assess the effectiveness and usability of the Interactive PyTamaro-Web Dashboard. The study aimed to establish whether the proposed pipeline could bridge the educational gap by converting raw telemetry data into meaningful information.

4.1 Evaluation Methodology

The validation was conducted through a digital survey hosted on the Qualtrics platform (see Appendix B for the complete questionnaire). The target audience consisted of instructors who actively utilize the PyTamaro-Web environment in their teaching routines. Participants were provided with a folder containing all the materials they needed to view and use the dashboard independently. In order to assess their interactions and determine whether the UI and data were consistent, user-friendly and easy to understand, they were asked to complete a set of predefined tasks. They were also given the option of recording their screen while navigating the dashboard. These optional recordings were intended to provide more in-depth information on user navigation and identify any usability issues.

The evaluation was structured based on specific scenarios that an instructor might deal with after a week of teaching, in order to simulate a standard workflow. The tasks were divided into two main categories:

4.1.1 Dashboard-Wide Tasks

This phase focused on high-level navigation and data aggregation. Participants were asked to:

- Navigate the global author list to find their profile.
- Sort their activities using the available tools, such as sorting by the highest number of users.
- Evaluate the “Curriculum List” view to see how well the dashboard summarizes metrics for grouped activities.
- Identify the specific activity that generated the highest user error rate.
- If it is possible, change the variant of one activity.

Immediately after completing each of these tasks, participants were asked open-ended questions to assess how easy and intuitive the interaction was, and whether the provided data felt helpful and actionable.

4.1.2 In-Depth Activity Tasks

Once they had successfully figured out the high-level metrics, participants were instructed to explore the dedicated “Activity Page” to test the dashboard’s core logic and insights. The tasks included:

- Identify the most frequent static errors (e.g. Invalid Syntax or Invalid Return Type) generated by users.
- Select an individual user from the drop-down menu, then use the provided UI controllers to navigate through their chronological execution timeline.
- See how the interface dynamically filters the displayed errors by switching between the different Static Error Analysis modes (None, Syntax and Syntax & Semantics).
- Compare the “Split” and “Diff” view modes to determine which one more effectively highlights the logical changes made by the user against the original instructor template.

As with the previous phase, participants answered open-ended questions after each task to provide qualitative feedback on the clarity, usefulness, and overall intuitiveness of the specific analytical tools tested.

4.2 Survey Design and Metrics

After exploring the dashboard in detail, the participants completed a questionnaire designed to capture both usability metrics and pedagogical feedback.

System Usability

The survey used a 5-point Likert scale ranging from “Strongly Disagree” to “Strongly Agree”. Participants were asked to evaluate the system based on:

- The ease of navigating the platform without prior training.
- The intuitive nature of the terminology and data displayed (e.g. bar charts, indices and dates).
- The effectiveness of the Activity Page in identifying why a user has failed an exercise.
- Their overall intention to integrate the interactive dashboard into their standard teaching routine.

Open-Ended Feedback

Participants were also asked open-ended questions about the pedagogical usefulness of the tool. This section focused on collecting feedback on whether the provided analytics support teaching decisions, and identify any valuable metrics missing. Finally, participants were encouraged to suggest improvements to the workflow and report any unexpected behaviors or bugs encountered during the testing phase.

4.3 Results and Discussion

The survey was completed by an instructor who creates and utilizes curricula on the PyTamaro-Web platform. The feedback provided valuable insights into the technical usability and pedagogical effectiveness of the dashboard.

4.3.1 System Usability Scale (SUS) Results

The responses to the System Usability Scale (SUS) section of the questionnaire were positive. The participant reported that the platform was easy to use even without prior training, and found the presentation of information, including bar charts, reports, and dates, to be intuitive.

The study confirmed the intended purpose of the dashboard. The activity page helped the participant understand why a user had failed and enabled them to identify the underlying cause. Furthermore, the participant stated that they would be pleased to incorporate the PyTamaro-Web interactive dashboard into their regular teaching routine. The only aspect that received less positive feedback was the intuitiveness of the terminology, highlighting areas where the language used in the interface could be improved.

4.3.2 Pedagogical Insights and Value

The instructor noted that the clean, minimalist design of the dashboard effectively highlighted critical information. On the Curriculum Page in particular, they found the “frequent errors” section more interesting and helpful for identifying the areas in which users struggle the most, rather than the total number of executions.

From a pedagogical perspective, instructor emphasized the importance of chronological run analysis. The ability to visualize the exact moment when a user transitions from making basic syntax errors during the initial coding phase to making more complex semantic errors provides important insight into their learning progression.

4.3.3 Constructive Feedback and Future Improvements

While the dashboard provides useful insights, qualitative feedback revealed some areas where technical and user experience improvements could be made. These suggestions can be grouped into three main categories:

Interface Mechanics and Navigation

One area for improvement is the execution viewer’s mechanics. The instructor noted that, when an error is clicked on in the analysis tab, the interface highlights the code line in question, but the screen does not automatically scroll down if the code block is lengthy. This forces to scroll manually to find the highlighted line. Furthermore, it was reported that navigating through users’ runs using long drop-down menus was cumbersome. To solve this issue, it was proposed that a more intuitive selection tool be implemented, allowing runs to be organized according to the number of errors in each run. Finally, some minor UI issues were reported, such as the inability to deselect a user from the dropdown menu without refreshing the page, and some ambiguity regarding specific UI elements, such as the “person icon”.

Data Clarity and Curriculum Alignment

The feedback highlighted the need for better contextualization of the data. The instructor had difficulty navigating the curriculum list because multiple curricula had the same name. There was also a discrepancy between the displayed participant counts and the expected class size, suggesting that the calculation of participant metrics needs to be clarified.

Code Review and Analysis Tools

The instructor should opt for the "Semantics only" filter over the "Syntax & Semantics" analysis mode, as users progress, syntax errors become less important and errors in semantics become more significant. Regarding the code comparison tools, the "Diff" view was found to be less helpful for exercises where the initial code template is almost empty (e.g. containing only ...). In these scenarios, users write new blocks of code instead of making minor changes, so the standard "Split" view is more appropriate. In addition, the instructor suggested that error cards could be improved by integrating references to function documentation and signatures.

5 Conclusion

Although interactive educational platforms such as PyTamaro-Web offer a hands-on learning environment, instructors can never be sure exactly how users achieve the final result. This thesis successfully bridges the gap between raw telemetry and useful insights by introducing the Interactive PyTamaro-Web Dashboard.

In order to achieve this objective, the project had to address several technical challenges related to both the integration of the large volume of collected data and the analysis of user-generated code. The developed pipeline, designed to operate entirely offline, successfully reconstructs anonymous user sessions by combining data stored in the database with information extracted from the notebooks contained in the original PyTamaro-Web repository. The introduction of a global execution cache significantly optimized backend processing, reducing the number of analyses required by the static analyzer `ty` from over 100,000 executions to approximately 76,000. Thanks to this optimization, the entire pipeline can process data from 13 authors in just 76 seconds. The generated frontend, built with Jinja2 and Tailwind CSS, provides a fully static, ready-to-use HTML interface, eliminating the need for an API-based backend and ensuring full offline portability. Furthermore, the integration of the Monaco editor, combined with the execution history, enables detailed comparisons between different versions of the code. This enables instructors to visually monitor users' learning progress.

Since supporting instructors in their teaching activities is the primary goal of this project, a usability study was conducted with an instructor who is currently using the platform. The study showed that the dashboard is an effective pedagogical and usability tool, enabling instructors to reconstruct user sessions and more easily identify issues that users are repeatedly encountering.

Future Work

Even though the current implementation provides a robust foundation for telemetry analysis, the validation study identified some areas that could be improved in the future.

- **Interface and Navigation Enhancements:** It was cumbersome to navigate extensive user histories through long dropdown menus. Future iterations should implement more intuitive selection tools, such as the option to filter runs by the number of errors. The execution viewer should also be improved so that it automatically scrolls to the highlighted line when an error card is clicked, reducing the need for manual navigation through long code blocks.
- **Data Contextualization:** Discrepancies in participant metrics and difficulties in distinguishing between curricula with the same name indicate a need for clearer and more aligned data, as well as a clearer understanding of expected class sizes.
- **Advanced Code Review Features:** The `Diff` view mode is currently less effective for activities where the initial template is almost empty. Additionally, providing direct references to function signatures and platform documentation on the error cards would help instructors to understand user mistakes more easily.

Overall, the Interactive PyTamaro-Web Dashboard turns an inaccessible telemetry database into a user-friendly analytical tool. This gives instructors the opportunity to refine their curricula over time and provide better, data-driven educational experiences.

References

[1] A. Altadmri and N. C. Brown. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 522–527, New York, NY, USA, Feb. 2015. Association for Computing Machinery. doi:10.1145/2676723.2677258.

[2] Astral. Ty. https://docs.astral.sh/ty/, 2026.

[3] L. Chiodini. *Teaching Introductory Programming Using Graphics as a Domain*. PhD thesis, Università della Svizzera italiana, Lugano, Switzerland, 2025. URL: https://susi.usi.ch/documents/332702/preview/2025INF011.pdf.

[4] L. Chiodini, J. Bevilacqua, and M. Hauswirth. The Toolbox of Functions: Teaching Code Reuse in Schools. In *Proceedings of the 6th European Conference on Software Engineering Education*, ECSEE '25, pages 185–189, New York, NY, USA, June 2025. Association for Computing Machinery. doi:10.1145/3723010.3723029.

[5] E. L. Glassman, J. Scott, R. Sing, P. J. Guo, and R. C. Miller. Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 2015. doi:10.1145/2699751.

[6] Lugano Computing Education research lab (LuCE). Pytamaro: Learn programming in python with graphics. https://pytamaro.si.usi.ch/.

[7] Microsoft. Monaco editor. https://microsoft.github.io/monaco-editor/.

[8] Pallets Projects. Click: Python composable command line interface toolkit. https://click.palletsprojects.com/.

[9] Pallets Projects. Jinja: A fast, expressive, extensible templating engine. https://jinja.palletsprojects.com/.

[10] Project Jupyter. Project jupyter. https://jupyter.org/.

[11] Tailwind Labs. Tailwind css: A utility-first css framework. https://tailwindcss.com/.

[12] The PostgreSQL Global Development Group. Postgresql documentation: Json functions and operators. https://www.postgresql.org/docs/current/functions-json.html.

[13] The PostgreSQL Global Development Group. Postgresql documentation: With queries (common table expressions). https://www.postgresql.org/docs/current/queries-with.html.

Appendices

A Statement on AI Contributions

During this project, generative AI tools were used to assist with coding, web design, and report writing.

Gemini

Name: Gemini (1.5 Pro)

Category: Conversational chatbot

Description: Gemini was utilized to streamline the research process, explain the documentation, and clarify some doubts about Python programming concepts. For the frontend, it provided UI layout suggestions. For the backend, it assisted in brainstorming session to improve execution efficiency and core logic.

Coverage:

- **Code: Backend (Python):** provided specific examples of Python libraries.
- **Code: Frontend (HTML/CSS/JS):** suggested UI layouts and suggested more efficient approaches for dynamic frontend behaviors.
- **Thesis: Entire Document:** it was used for grammar and syntax checking to ensure correct English wording.

B Interactive Dashboard Usability Survey

This appendix contains the full survey that was used to validate the Interactive PyTamaro-Web Dashboard. It was designed to collect feedback from instructors about their experience.
The digital version of the survey was hosted on the Qualtrics platform and could be accessed via the following link: `https://usi.qualtrics.com/jfe/form/SV_eET1G27CigsUY0C`
The original survey document is provided in full on the following pages.

Introduction & Welcome

Please read this before you start.

This survey aims to evaluate the new **Interactive PyTamaro-Web Dashboard** that has recently been developed for the PyTamaro-Web platform. This project aims to provide an in-depth analysis of data relating to your teaching activities, helping you to monitor users' progress and identify common errors.

I acknowledge and agree that any information or materials provided to me in order to participate in this survey are confidential and must not be shared with anyone else.

◯ I agree ◯ I disagree

Please fill out this form before starting the survey.

Name and Surname: _____

Email: _____

I consent to being contacted via email in case there are any questions or doubts about my survey responses.

◯ I agree ◯ I disagree

This survey proposes a few tasks using the generated **Dashboard** to collect your feedback. Specifically, you will:

- Navigate the website.
- Search for activities and curricula.
- Review the user runs.
- Explore other features within the Dashboard.

Optional Screen Recording
It would be helpful to record your screen while you complete these tasks, though **this is not required**. This allows us to see exactly how you interact with the interface and helps us to identify which features are easy or difficult to find. If you choose to record, please try to keep both the **Dashboard** and this **Qualtrics** survey visible on your screen at the same time. This will help us to accurately connect your actions with your written feedback.

Dashboard-Wide Tasks

*Scenario: You have just finished teaching for the week and want to evaluate the class's overall performance using the **Dashboard**.*

General Overview
*You are currently on a page displaying a table of activities authors, but in this scenario, only your name is listed. Please click on your name to view a list of all your activities and their metrics. Next, find the sorting dropdown and sort your activities by **Most Users**. Finally, take a look at the individual information blocks provided for each activity.*

Looking at how the data is grouped for each activity, what are your first impressions of the layout and the information provided?

Curriculum Assessment
*Now you are currently in the **Author Dashboard** page, please navigate to the **Curriculum List** and select a curriculum of your choice. Take a brief look at the information and metrics summarized for the activities listed there. Next, click on any activity to open its dedicated page.*
*(**Note:** You can explore this activity page in more detail later, there is a dedicated task section for activities)*

How easy was it to locate a curriculum? And looking at the numbers provided for the activities, did they make sense to you?

Identifying User Difficulties
*Please return to the **Author Dashboard** page and navigate to the **Activity List**. From there, try to find out which specific activity had the **highest error rate** among your users.*

Overall, how easy was it to move around the website? Did you feel it was obvious how to sort the list and locate that specific activity?

Switching Between Versions (If possible)
If possible, please find or search for an activity in the list that has at least two versions. Once you have found one, try switching between the versions directly from this screen without opening the activity to view more details.

Did the website make it easy to switch between versions? Did you encounter any issues?

In-Depth Activity Page Tasks

Follow the step-by-step instructions below to test the detailed features, including the code review tools.

Select Activity

1. If you are not on the right page, navigate to the **Activity List** in the **Author Dashboard**.
2. Select an activity from the available list.

Which activity did you select?

Detailed Activity Analysis

1. Identify the **most frequent error type** (e.g., Invalid Syntax) made by your users.
2. Click on the error name. This will take you to a page on the website that explains exactly what the error means.

What was the most frequent error type you identified? How easy was it to locate this information using the error types pane, and did you find the detailed explanation page useful?

Individual User Metric

1. Select one of your users.
2. Look at the **Runs/Cell** metric next to the label **Select Run Timeline**.

Which user did you choose? What does the **Runs/Cell** indicator mean to you?

Run Navigation

1. Use the arrows below the **Select Run Timeline** label to navigate between runs and observe how the analysis column and the user code column changes over time.

Is it intuitive to navigate between runs?

Analysis Mode

1. Select one of your users, if none is currently selected.
2. Look at the **Static Error Analysis** mode and hover the information icon to read the description.
3. Select an execution that contains at least one more error in **Syntax & Semantics** than in **Syntax**.
4. Switch between the analysis modes and observe the **analysis** column.

Which user did you choose? Were you able to identify the user's default selection? Does selecting an analysis mode help to filter out errors? Is the UI intuitive?

Code Review

1. Select one of your users, if none is currently selected.
2. Then select one of your users and choose the **Syntax & Semantics** analysis mode.
3. Navigate between the runs and look at the **Analysis** column.
4. When you find an interesting error, click it to highlight the corresponding line, look at the **User Cell** and try to understand the meaning of the error.

Which user and run did you choose? Is the analysis column meaningful and helpful? Do you find the feature that highlights error lines useful?

Diff View

1. Select one of your users and choose the **Syntax & Semantics** analysis mode.
2. Switch to the **Diff** view mode and navigate through the different runs.
3. Examine the **Diff: Initial vs User Cell** to understand what the user changed compared to the **Initial Code**.
4. Return to the **Split View** mode and try to identify the same changes you observed in **Diff** mode.

Which user did you choose? Is the **Diff** view helpful? Which view mode do you find more effective for identifying users' changes and their code logic, and why?

Open Exploration

You're almost done, this is the final step. Take a few minutes to explore the platform at your own pace, just as you would during a normal working day. Feel free to experiment and explore anything that interests you. Test the system's limits.

Post-Task Questionnaire

General feedback and usability ratings.

A - SYSTEM USABILITY

	Strongly Disagree	**Disagree**	**Neither Agree nor Disagree**	**Agree**	**Strongly Agree**
1. I found the platform easy to navigate, even without prior training.	○	○	○	○	○
2. The terminology is intuitive.	○	○	○	○	○
3. The way the data is visualised (bars, ratios, dates) is intuitive.	○	○	○	○	○
4. The Activity Page helped me to understand why a user had failed and to deduce their reasoning.	○	○	○	○	○
5. I would be happy to use Interactive PyTamaro-Web Dashboard in my teaching routine.	○	○	○	○	○

B - OPEN-ENDED FEEDBACK

1. Pedagogical Usefulness
Does this data help you to make teaching decisions? Which analytics or metrics are currently missing that would be highly valuable?

---

2. Workflow Improvements
What changes or additions would you make to this tool to make it fit your workflow better?

---

3. Overall Feedback
Did you find any features that you liked? Were there any unexpected behaviour, bugs or dead ends?

---

4. Additional Suggestions
Do you have any other suggestions regarding the entire project?

Screen Recording Link
If you recorded your screen to improve validation, please provide the link or upload the recording.
If the upload fails due to file size limitations, please send the recording separately via WeTransfer (or a similar service) to: `simone.maccario04@gmail.com`
