information

MDPI

Article

A Novice-Friendly Answer Interface with Code Behavior Visualization and AI Assistant for a Python Programming Learning Assistant System

Zhida Fu [1], Nobuo Funabiki [1,*], Zihao Zhu [1], Yue Zhang [1], Wen-Chung Kao [2], Yi-Fang Lee [3] and Pi-Kuang Tseng [3]

[1] Department of Information and Communication Systems, Okayama University, Okayama 700-8530, Japan; pb6n33wm@s.okayama-u.ac.jp (Z.F.); pjco5fyn@s.okayama-u.ac.jp (Z.Z.); pr1u5yrh@s.okayama-u.ac.jp (Y.Z.)
[2] Department of Electrical Engineering, National Taiwan Normal University, Taipei 106308, Taiwan; jungkao@ntnu.edu.tw
[3] Department of Industrial Education, National Taiwan Normal University, Taipei 106308, Taiwan; ivana@ntnu.edu.tw (Y.-F.L.); tbc0147@ntnu.edu.tw (P.-K.T.)
* Correspondence: funabiki@okayama-u.ac.jp

Abstract

Nowadays, *Python* is very popular as the first programming language for novices, including high school students, to learn due to its short code features with rich libraries. Thus, it is important to provide a learning environment supporting studies starting from the fundamentals, since students have no knowledge on how a program runs on a computer. Previously, we have developed a web-based *programming learning assistant system (PLAS)* to allow the self-study of major programming languages, including *Python*, by university students. It offers several types of exercise problems that have different learning goals and levels for step-by-step study. Any student answer is automatically marked at the answer interface for quick feedback. However, *PLAS* has not implemented functions to assist the learning needs of high school-level students. In this paper, we propose a novice-friendly answer interface for a *Python programming learning assistant system (PyPLAS)* that introduces a code behavior visualization and an AI assistant with learning logs. The visualization allows learners to observe the changes in variable states and the control flow. The assistant provides multi-level hints during learning and reflective feedback after it by analyzing the logs based on engagement, reasoning strategies, learning pace, and tool usage. For evaluation, we implemented the proposed interface using *Python Flask* for the web platform and *Ollama* as a locally deployed AI model. A pilot application was conducted with high school students solving introductory Python exercises in *PyPLAS*. The results showed high task completion, positive questionnaire responses toward embedded visualization and interface usability, and teacher-observed usefulness of the four-dimensional learning analytics for interpreting learner behaviors. These findings provide preliminary evidence for the feasibility and practical value of the proposed interface, while larger controlled studies are required to validate its instructional effectiveness.

Keywords: Python programming learning assistant system (PyPLAS); novice-friendly; programming learning environment; program visualization; AI-assisted learning; learning analytics

Check for updates

Academic Editor: Jui-long Hung

Received: 8 April 2026
Revised: 15 May 2026
Accepted: 18 May 2026
Published: 21 May 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

Nowadays, *Python* is widely adopted as a first programming language in introductory computing education due to its readable syntax, interpreted execution model, and rich

libraries [1,2]. For high school students and other novice learners, however, practical learning difficulty remains high. Learners may observe final outputs, whereas they cannot directly observe variable transitions, branching decisions, loop states, or function call behavior during execution. This invisibility often leads to repetitive trial and error code edits, which will be a barrier for the construction of stable mental models.

As an ongoing project, we developed the web-based *programming learning assistant system (PLAS)* to support the self-studies of novices through structured exercises and automatic answer checking [3,4]. *PLAS* has been applied in university-level programming courses for popular languages such as *C*, *Java*, *Python*, and *JavaScript*. Nevertheless, its current design does not fully support high-school level students learning *Python*. It lacks intuitive mechanisms to reveal internal execution behaviors within the answer interface and to process data for providing adaptive pedagogical feedback.

In this paper, to address these gaps, we propose an answer interface that integrates code behavior visualization, event stream behavioral logging, and AI-assisted feedback within a unified learning workflow for *Python programming learning assistant system (PyPLAS)*. During problem solving, learners receive progressive hints at multiple levels designed to support reasoning without directly revealing final answers. At chapter checkpoints, learners can request reflective feedback from the submission section. The system analyzes learning logs across the following four dimensions: engagement, reasoning strategies, learning pace, and tool usage. Based on the analysis result, it generates reflective outputs to support learner awareness of their learning processes.

The proposal forms an integrated learning loop within a single interface and service pipeline. It combines code execution visibility, process logging, staged AI hinting, and reflective analytics at a chapter level. It introduces a novice-friendly answer interface that embeds the stepwise code execution visualization directly into the primary practice workflow, enabling learners to observe program behaviors during problem solving. It also records detailed learner interactions through an event stream logging framework and analyzes the logs to understand learning behaviors of learners. Based on this information, it provides AI assistance in two stages as follows: progressive hints during problem solving and reflective feedback requested at chapter checkpoints. The interface is implemented using a *Flask*-based backend [5] together with locally deployed large language model services *Ollama* [6], allowing practical deployments in classroom environments with limited infrastructure. In preliminary pilot use, high school teachers and students solved introductory *PyPLAS* exercises with the proposed interface, and the results suggest that this integration may help reduce excessive trial-and-error behaviors in this pilot context.

The objective of this study is to design and evaluate a novice-friendly *PyPLAS* answer interface that integrates code behavior visualization, event-stream learning logs, and AI-assisted feedback for high school students who are learning introductory *Python* programming. This study examines whether the proposed interface makes the program execution process more visible, provides staged learning support, and generates interpretable learning analytics for classroom use. Since the present study is positioned as a pilot application, our evaluation focuses on technical feasibility, preliminary usability, and practical educational value rather than on establishing causal effectiveness through a controlled experiment.

To guide the design and evaluation, this study addresses the following research questions:

- **RQ1:** Does the integrated visualization improve learner understanding of runtime behavior?
- **RQ2:** Does the progressive AI hint help limiting excessive trial-and-error attempts of answer submissions?

- **RQ3:** Is the four-dimensional learning analytics useful for pedagogical interpretation?
- **RQ4:** Is the implemented interface usable and accepted by high school students?

The rest of this paper is organized as follows. Section 2 discusses features of the proposed answer interface. Section 3 presents the design concept of the answer interface for *PyPLAS*. Section 4 describes the system design and implementation. Section 5 reports the evaluation setup and preliminary findings. Finally, Section 6 concludes the paper with future work.

2. Features of Proposal

In this section, we discuss pedagogical rationale and related work underlying the proposed answer interface for *PyPLAS*, focusing on programming learning platforms, code visualization, learning analytics, and AI-assisted programming education.

The proposed interface was not intended merely as a technical combination of code visualization, AI assistance, and learning logs. Instead, their combination was properly designed to address three major learning difficulties faced by novice programmers: difficulty in understanding invisible runtime behavior, dependence on blind trial and error, and limited awareness of learning strategies. Accordingly, code behavior visualization is adopted in relation to active learning and mental model construction, progressive AI hints are adopted as staged support informed by formative feedback, and learning logs are adopted in relation to self-regulated learning and learning analytics.

2.1. Code Visualization for Beginners

Code visualization is recognized as a practical bridge between static source code and dynamic execution semantics [7,8]. Beginners may frequently misunderstand variable updates, branch selections, loop state transitions, and function call scopes because these operations are invisible during normal execution. Visualization tools can reduce this invisibility by externalizing the execution state and control flow [7].

As a popular visualization tool, *Python Tutor* [7] can be used externally. Although it is effective in helping high school students grasp basic grammar concepts, it revealed a critical usability issue that learners must manually copy code back and forth. This context switching increases cognitive overhead, which makes many learners stop using this visualizer.

To address this, visualization should be tightly coupled with the problem solving interface [9]. Therefore, in this paper, we embed *Python Tutor* into the practice page. This direct integration ensures that learners can form explicit predictions, run stepwise traces, and revise their mental models without leaving the learning environment.

Recent platform-based approaches have also emphasized the importance of integrating practice environments, automated feedback, and teacher support in programming education [10,11].These studies show that programming learning platforms are increasingly expected to support not only answer checking but also instructional management and feedback delivery. Compared with these systems, our proposed *PyPLAS* interface focuses on high school *Python* programming beginners and integrates runtime visualization directly into the answer interface.

From a pedagogical perspective, embedded visualization supports active learning where learners are encouraged to inspect, predict, and verify program behavior during problem solving rather than observing the final output only [12]. For novice programmers, understanding code requires the construction of mental models about how a source code is executed step by step. By externalizing variable transitions, branch selections, loop iterations, and function call behavior, this visualization helps learners compare their expectations with actual execution traces and revise their understanding of runtime behavior.

2.2. Learning Analytics with Behavioral Logs

Learning analytics in programming education has shifted from outcome-based measurements such as final scores and pass/fail rates to process-aware measurements of interaction sequences, timing patterns, and help-seeking behaviors [13,14]. Process data can provide richer evidence for identifying learning strategies, confusion states, and self-regulation patterns.

Event-stream logging is particularly suitable for this purpose because it preserves the temporal order of actions and supports flexible re-aggregation [15]. Instead of merely storing final code snapshots, event streams allow the reconstruction of complete problem solving trajectories of the timings when a learner edits code, runs tests, requests hints, or opens the visualization tool. This granularity is crucial, especially for high school students, where teachers require interpretable process signals to provide targeted interventions.

A major challenge in process analytics, however, is pedagogical interpretability. Metrics that are mathematically complex but instructionally opaque are difficult for educators to apply in real classroom settings [13]. To ensure practical usability, we adopt an analytics model that distills complex event logs into four highly explainable dimensions—engagement, inference strategy, learning pace, and tool use—enabling consistent and objective analysis across learners.

This analytics design is related to self-regulated learning because it helps learners and teachers reflect on the learning process rather than relying on final correctness. The four dimensions of engagement, reasoning strategy, learning pace, and tool usage provide interpretable indicators for monitoring learning behaviors and adjusting subsequent learning strategies.

2.3. AI-Assisted Programming Education

AI support in programming education has evolved significantly, moving from traditional rule-based tutoring to interactive assistants driven by *large language models (LLMs)* [16,17].

Recent studies have also examined the use of generative AI in programming-related learning and source-code tasks, showing both its potential benefits and risks [18]. These findings support the need for careful instructional design when applying AI to programming education. Therefore, the proposed *PyPLAS* assistant adopts a non-replacement L1/L2/L3 hint policy rather than directly providing complete code solutions.

From an instructional perspective, contemporary AI-assisting programming systems generally employ progressive scaffolding and adaptive diagnosis [19]. However, prior reports highlighted a common failure mode in *LLM* applications: high convenience can lead to low cognitive engagement [20]. If an AI assistant provides complete code solutions too early, learners often skip the essential hypothesis formation and verification steps. This risk of excessive reliance is particularly critical in high school classrooms, where learners have not yet developed stable programming schemas.

Consequently, we emphasize conceptual and strategic guidance before offering code-level hints, strictly preventing direct-answer generation. Furthermore, while many AI tutoring systems operate independently of detailed learner trajectories or rely on cloud-based architectures that may pose privacy and stability issues in schools [21], our *PyPLAS* feeds the locally collected event stream metrics into a locally deployed *LLM*. This creates an adaptive, behavior-aware reflection mechanism that is both pedagogically safe and practically deployable.

In this study, the AI assistant is designed as a staged support mechanism informed by formative feedback. The purpose of the L1/L2/L3 hint policy is not to replace learners' reasoning or directly provide final code answers, but it is to guide learners during problem solving. Conceptual hints, strategy-level hints, and error-focused clues are provided progressively so that learners can continue diagnosing their own mistakes and refining understanding.

This design is also consistent with prior work showing that structured scaffolding can support novice programmers' comprehension of program behavior and problem meaning [22].

2.4. Integration of Pedagogical Rationale and Evaluation Focus

Taken together, the three components of the proposed interface correspond to the major learning difficulties identified in introductory *Python* programming. Code behavior visualization addresses hidden runtime behavior by supporting active observation and mental model construction. Progressive AI hints address blind trial-and-error behavior by providing formative, staged support during problem solving. Event-stream logs and four-dimensional analytics address limited metacognitive awareness by supporting reflection on engagement, reasoning strategy, learning pace, and tool usage. These pedagogical connections will guide the evaluation design of this study. Visualization was examined through learners' perceived understanding of runtime behavior. AI hints were examined through attempt and hint-use patterns. Learning analytics were examined through teacher interpretation and learner reflection. The integrated interface was examined through usability and acceptance.

3. Design Concept

In this section, we present the design concept of the answer interface for *PyPLAS*.

3.1. PLAS for University Students

For university students, *programming learning assistant system (PLAS)* has been developed as a web-based self-learning platform of programming in prior studies [3,4]. *PLAS* supports structured topic-based practices with the browser access, teacher-authored problems, and immediate answer checking. The exercise ecosystem includes both *code reading* and *code writing* tasks to support progressive step-by-step self-learning [3,4], which can be effective for university students with a basic programming background.

3.2. PLAS for High School Students

For high school students or absolute beginners in *Python* who are the targets of this study, instructional needs can be shifted toward runtime process interpretation and guided programming practices. *PyPLAS* needs a novice-focused extension that integrates execution visualization, staged AI hinting, and reflection at each programming topic using learning logs into one workflow.

Table 1 shows a comparison between *PLAS* for university and high school students.

Table 1. Comparison of *PyPLAS* for universities and high schools.

Aspect	University	High School
Target learners	novice to intermediate learners in universities	learners in high schools and absolute beginners of Python
Core practice mechanism	topic-based exercises with immediate answer marking	plus process-aware support during and after tasks
Execution-behavior support	no visualization	embedded visualization
AI tutoring support	no staged hint policy	progressive L1/L2/L3 hint policy
Learning analytics	activity records for learning history tracking	event stream analytics with four-dimensional topic reflection
Deployment style	web-based platform	Flask + Ollama deployment for classroom feasibility

3.3. Learning Challenges and Design Requirements

Based on observations from introductory *PyPLAS* deployments, five recurring challenges should be emphasized. They include (1) hidden runtime behavior, (2) trial and error dependence, (3) abstract concept barriers, (4) weak error interpretation, and (5) limited metacognitive awareness. These challenges are interconnected. Learners who cannot interpret execution states tend to over-rely on repeated reruns. Learners without strategy awareness tend to request help without diagnosing the root cause.

Accordingly, our *PyPLAS* extension adopts five operational requirements: (1) it records explainable interaction traces with timestamps and context, (2) it provides low-friction visual support directly in the solving interface, (3) it enforces non-replacement AI assistance that guides reasoning rather than giving final answers, (4) it generates reflective feedback at the chapter level from behavior logs, and (5) it is deployable into classrooms under constrained infrastructure.

The design requirements are connected to the pedagogical rationale discussed in Section 2. The requirement for low-friction visual support addresses the difficulty of understanding hidden runtime behavior by allowing learners to observe the execution process within the same problem-solving interface. The requirement for non-replacement AI assistance addresses the trial-and-error dependence and weak error interpretation ability by providing staged guidance without replacing the learners' reasoning. The requirement for explainable logging and chapter-level reflection addresses limited metacognitive awareness by making learning behaviors visible and interpretable for both learners and teachers. In this way, each design requirement is directly linked to a specific learning difficulty and to a corresponding support function in the proposed interface.

3.4. Instructional Workflow Perspective

Basically, the learning loop of *PLAS* consists of reading a question, submitting an answer, receiving correctness feedback, and retrying answer submission. This loop may be insufficient for absolute beginners who need support in interpreting runtime behavior and adjusting learning strategies.

Therefore, this extension of *PyPLAS* introduces a dual-loop instructional workflow. The *micro-loop* operates within each task and follows answer-input/modification-run-observe-hint-revise iterations. The *macro-loop* operates at the programming topic level and follows aggregate–analyze–reflect–plan transitions. This structure enables learners to improve local correctness during tasks while also improving learning strategy across tasks.

This dual-loop workflow clarifies how the proposed design is evaluated. The *micro-loop* focuses on task-level learning support, including the learners' use of embedded visualization and progressive AI hints during problem solving. It is related to the evaluation of

runtime-behavior understanding, attempt patterns, and hint-use patterns. The *macro-loop* focuses on topic-level reflection based on accumulated learning logs. It is related to the evaluation of four-dimensional learning analytics, teacher interpretation, learner reflection, and overall usability of the integrated interface.

4. Implementation

In this section, we present the implementation of the answer interface for *PyPLAS*.

4.1. Software Architecture

The extended *PyPLAS* adopts a lightweight local-first architecture with four layers as follows: presentation, application, AI service, and data/logging. Figure 1 shows the architecture-level organization of these layers. Table 2 summarizes the technologies adopted in each layer and their corresponding functional roles in the proposed system.

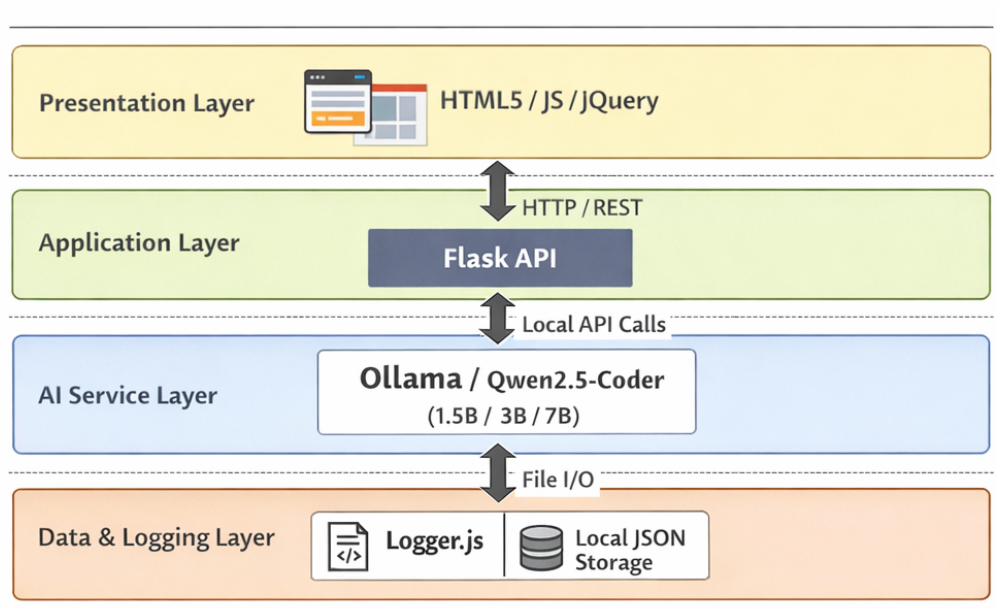

Figure 1. Software architecture of extended *PyPLAS*.

Table 2. Technical stack and functional roles.

Layer	Technology	Role in This System
Presentation	HTML5, JavaScript (ES6), jQuery	problem pages, editor interaction, AI panel, visualizer controls
Logging	author-developed `logger.js` module	event stream capture for user operations and context data
Application	Flask API	authentication, problem service, aggregation, AI request orchestration
AI service	Ollama + Qwen2.5-Coder (1.5 B, 3 B, 7 B)	local hint generation and chapter-end feedback generation
Data	local JSON-based persistence	session state, user progress, exported logs

The layer roles are summarized as follows.

- Presentation layer: The frontend is designed for browser-only access without additional plugins, and the interface prioritizes novice readability with low interaction complexity.

- Application layer: *Flask* handles RESTful endpoints for problem retrieval, answer submission, log ingestion, and AI request routing, while session state preserves chapter progress and contextual history.
- AI service layer: *Ollama* runs local *LLM* inference on classroom PCs. The default option is *Qwen2.5-Coder 3 B* [23] for balancing quality and latency, with 1.5 B and 7 B alternatives for lower and higher resource conditions.
- Visualization support: The interface embeds *Python Tutor* [7] through an *iframe* in the practice interface and uses it as an auxiliary visualization tool for step-by-step execution tracing.
- Data and export layer: Raw events are stored in append-only form and can be exported as *JSON* and plain-text summaries for teaching review and research analysis.

4.2. Operational Cycle

Figure 2 shows the operational cycle of basic *Python* study using the proposed answer interface. The operational cycle contains user interaction, behavior logging, AI response generation, and UI delivery. The details of each stage are as follows:

- Event data collection: User actions such as answer input/modification, code run, submission, opening visualizer, requesting hint, and page switching are captured with the timestamp, user ID hash, chapter/problem identifiers, and local context.
- Data aggregation: Behavioral features are aggregated at configurable boundaries (e.g., per problem and per chapter), with aggregation at the chapter level used for reflection triggering.
- AI feedback generation: For each request, the backend constructs a structured prompt that includes the learner's context, latest answer/error status, recent behavior features and required response type (L1/L2/L3 hint or chapter reflection).
- Feedback delivery: Hints are shown in the AI panel during tasks, while chapter reflection is delivered in a modal window with visual summary and textual suggestions.

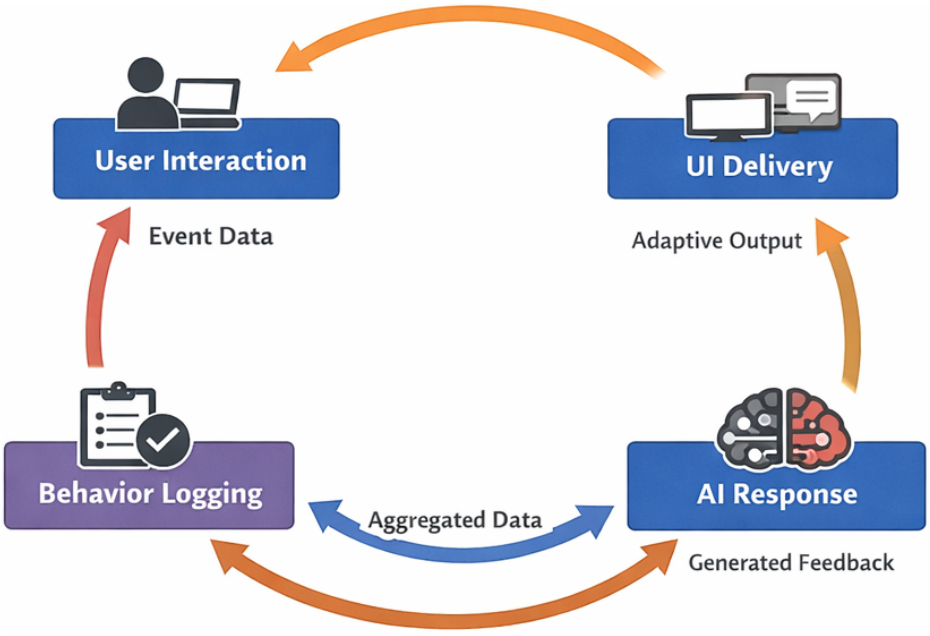

Figure 2. Operational cycle of proposal.

Figure 3 provides an end-to-end view of how these components are connected in the learning workflow.

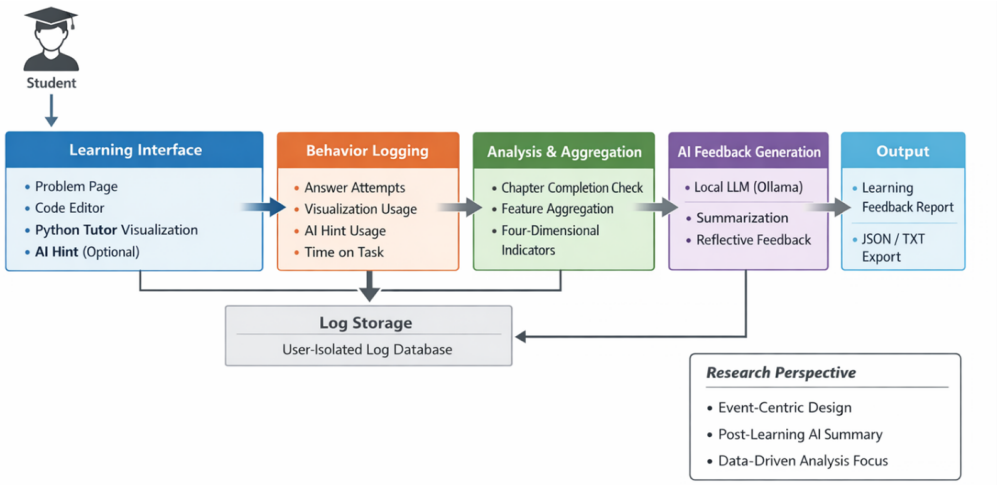

Figure 3. End-to-end learning workflow.

4.3. Four-Dimensional Learning Analysis

To analyze learning behaviors from raw event-stream logs, *PyPLAS* adopts a four-dimensional learning analysis model. This model is intended to summarize learner behaviors in an explainable form for classroom use rather than to provide a fully automated diagnosis of learner ability.

The four dimensions are selected by three reasons. First, they can be computed directly from event-stream logs without questionnaires or manual annotations. Second, they correspond to behavior patterns that teachers can interpret in high school programming classes. Third, they cover the main aspects of learner behaviors in learning *PyPLAS*. The model consists of engagement, reasoning strategy, learning pace, and tool usage. The current parameters for them are selected in a heuristic way where they should be refined through larger-scale data in future work.

4.3.1. Engagement

Engagement represents the extent to which a learner remains actively involved in assigned learning activities. In high school programming practice, weak engagement often appears as short active time, incomplete task coverage, or early disengagement from practice. To reflect both temporal participation and completion behavior, *Engagement* is computed from active learning time and task completion coverage:

$$E = \alpha \cdot \frac{T_{\text{active}}}{T_{\text{target}}} + (1 - \alpha) \cdot \frac{N_{\text{complete}}}{N_{\text{assigned}}}. \tag{1}$$

Here, $\alpha \in [0, 1]$ is a weighting coefficient that balances time investment and task completion. In this paper, $\alpha = 0.5$ is used to weight both factors equally. A lower value of E suggests insufficient participation or incomplete progress within the assigned learning unit.

4.3.2. Reasoning Strategy

Reasoning strategy estimates how learners use visualization and answer submissions during problem solving. In *PyPLAS*, it can be observed through the balance between visualization use and answer submission behavior. Therefore, the following ratio is defined:

$$R_{\text{val}} = \frac{\text{Vis_Open_Count}}{\text{Vis_Open_Count} + \text{Answer_Submit_Count}}. \tag{2}$$

In the current pilot setting, a relatively high R_{val} indicates the frequent use of execution tracing before submission, whereas its relatively low value suggests a tendency toward

direct submission without sufficient use of visualization. This indicator is used to analyze reasoning behavior during problem solving.

4.3.3. Learning Pace

Learning pace captures the temporal stability of learner actions during practice. For high school students, highly irregular action intervals may reflect fragmented trial and error behavior, confusion, or unstable self-regulation, whereas a more stable rhythm may indicate a more controlled problem solving pattern. In this paper, *learning pace* is modeled by the coefficient of variation of inter-action intervals:

$$CV_{\text{val}} = \frac{\sigma(\Delta t)}{\mu(\Delta t)}. \tag{3}$$

Lower values indicate a relatively stable learning rhythm, while higher values suggest more fluctuating action timing. The threshold used in this pilot study is heuristic and intended for tentative classroom interpretation.

4.3.4. Tool Usage

Tool usage captures the extent to which learners used AI-assisted support during problem solving. Since *PyPLAS* provides staged L1/L2/L3 hints, help-seeking behavior can be represented by the frequency of hint requests relative to problem-solving attempts:

$$U_{\text{ratio}} = \frac{N_{\text{L1}} + N_{\text{L2}} + N_{\text{L3}}}{N_{\text{attempt}}}. \tag{4}$$

A higher U_{ratio} indicates more frequent use of hint support. When interpreted together with the other dimensions, this indicator helps analyze how learners use external support during problem solving.

4.3.5. Integration of Four Dimensions

The proposed system interprets the four dimensions jointly. For example, high engagement with stable learning pace may indicate persistent and organized study, whereas high engagement with highly fluctuating pace may suggest effortful but fragmented learning. Similarly, frequent visualization use together with moderate hint use may reflect active reasoning, while low visualization use with frequent high-level hints may suggest dependence on external support. This interpretation is used to support teacher reflection and learner awareness at the chapter level.

4.4. User Interface

The user interface of *PyPLAS* consists of the following pages.

4.4.1. Login Page

Login page provides the student-ID-based access and initializes a local learning session for each learner. Figure 4 shows the entry page interface.

4.4.2. Problem List Page

The *problem list page* organizes exercises by chapters and problem numbers, and it provides a submission section for requesting AI feedback and exporting log files after practice. Figures 5 and 6 show the upper and lower portions of this page.

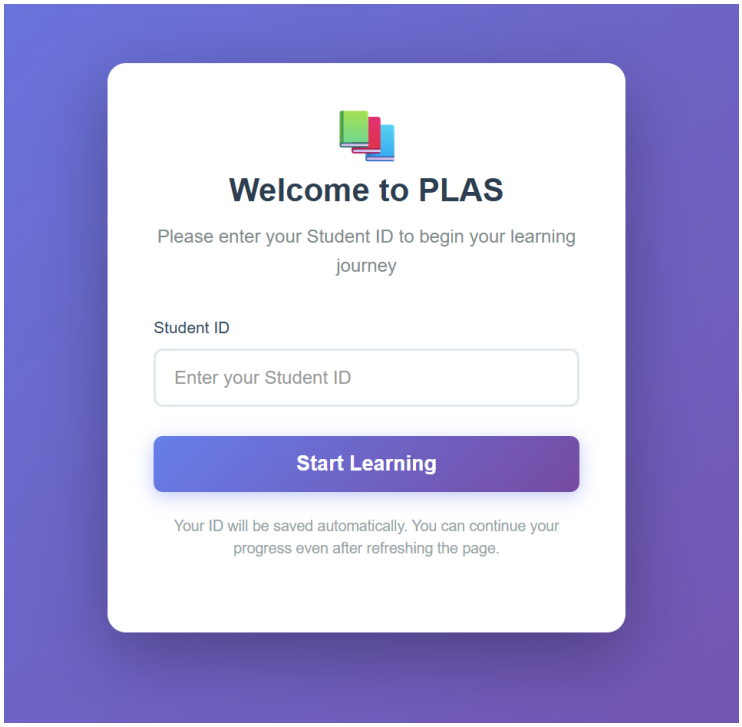

Figure 4. Login page of *PyPLAS*.

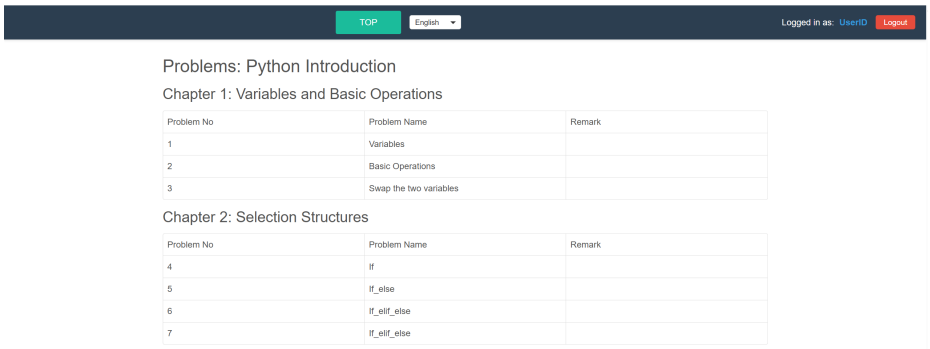

Figure 5. Problem list page (upper section).

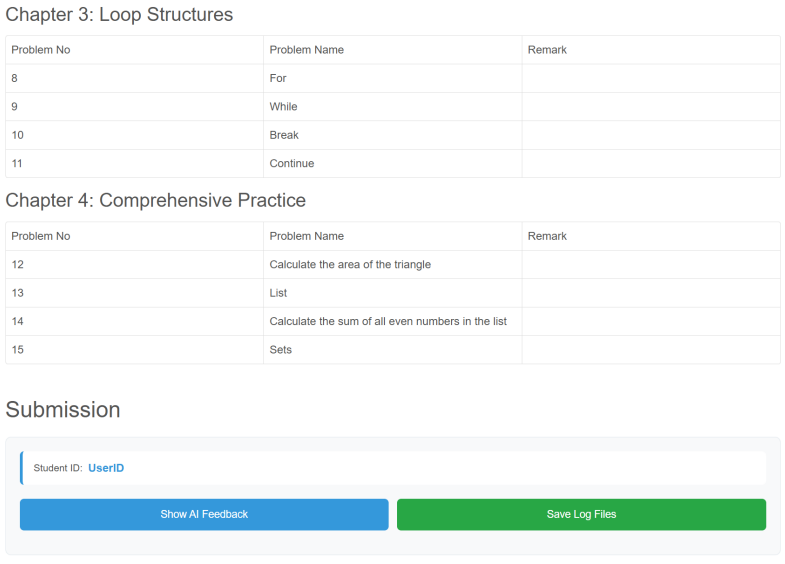

Figure 6. Problem list page (lower section).

4.4.3. Problem Practice Page

The *practice page* integrates the problem statement, code display, answer input fields, AI assistant panel, and access to the embedded *Python Tutor* visualization within a single interface. Learners can open *Python Tutor* to inspect step-by-step execution and verify intermediate variable states during problem solving. Figure 7 shows the integrated problem practice interface.

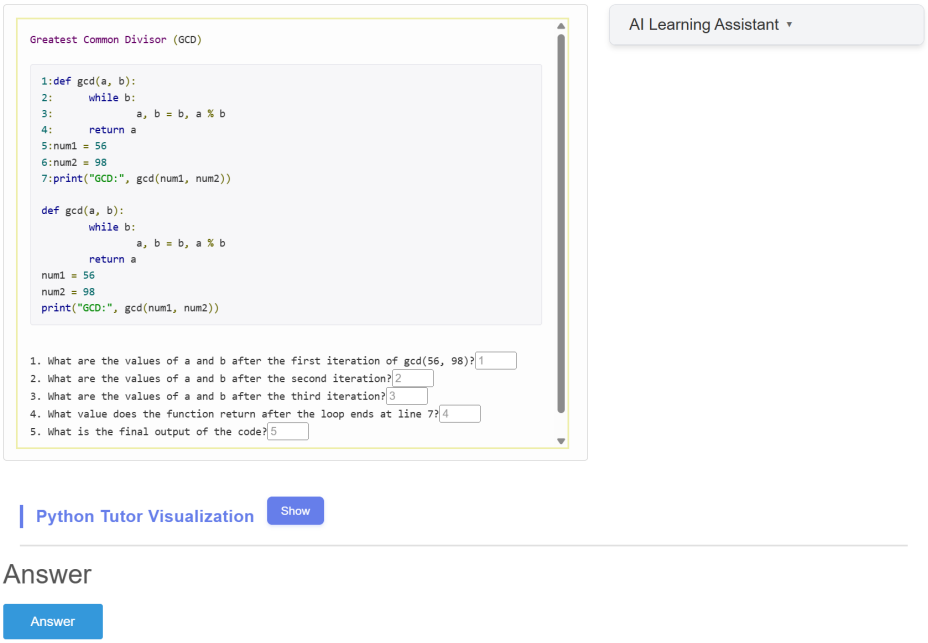

Figure 7. Integrated problem practice interface.

The AI panel provides progressive hints at the following three levels:

- Level 1: Conceptual guidance.
- Level 2: Problem-solving strategy.
- Level 3: Error-focused guidance and correction direction.

Figure 8 illustrates the hint interface, where learners can first view a conceptual hint and then can request a higher-level hint when needed.

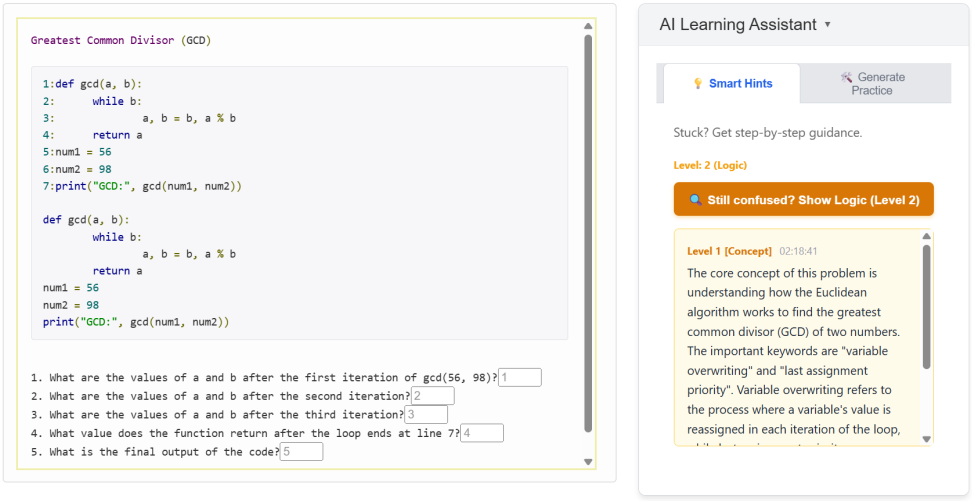

Figure 8. AI assistant panel showing staged hint delivery.

Figures 9 and 10 show the embedded visualization views before execution and during execution tracing, respectively.

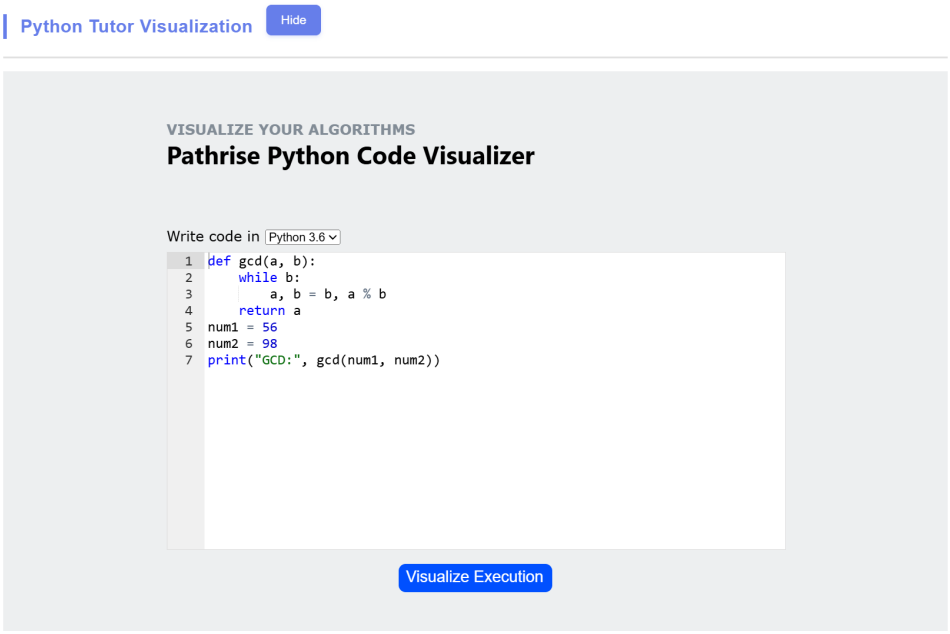

Figure 9. Embedded *Python Tutor* view before execution.

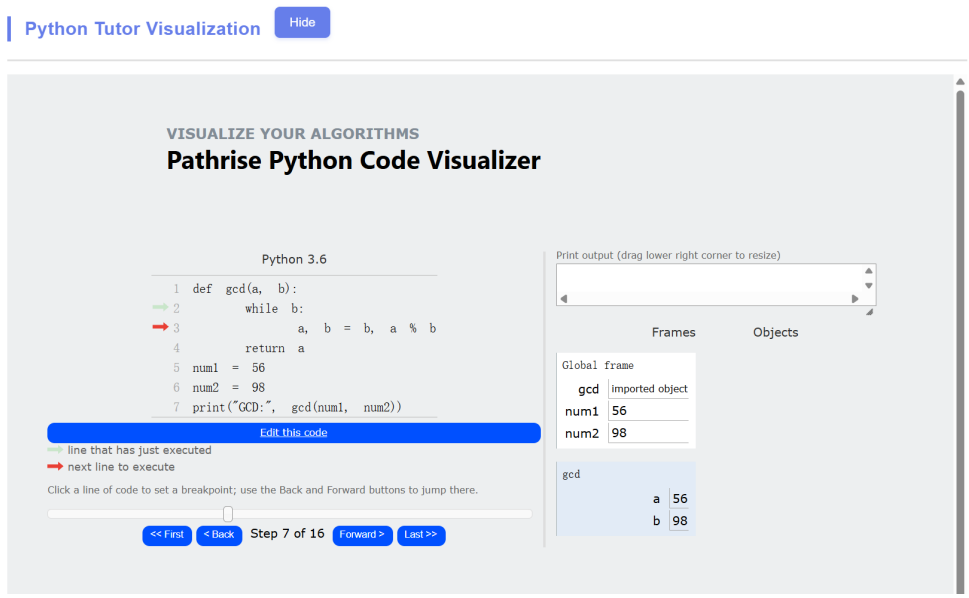

Figure 10. *Python Tutor* execution-trace view.

4.4.4. AI Feedback Modal Window

After chapter-level practice, learners can click the AI feedback request button in the submission section to open a reflection modal that summarizes learner behavior, presents dimension-wise interpretations, and provides reflection questions for metacognitive support. Figure 11 shows an example of the chapter-end AI learning diagnosis report.

4.4.5. Log Export Functionality

PyPLAS supports the log export in *JSON* and text-summary formats for self-reflection, teacher review, and research use. After an export is completed, the interface confirms the generated file names and upload status, as shown in Figure 12.



Figure 11. Chapter-end AI learning diagnosis report.

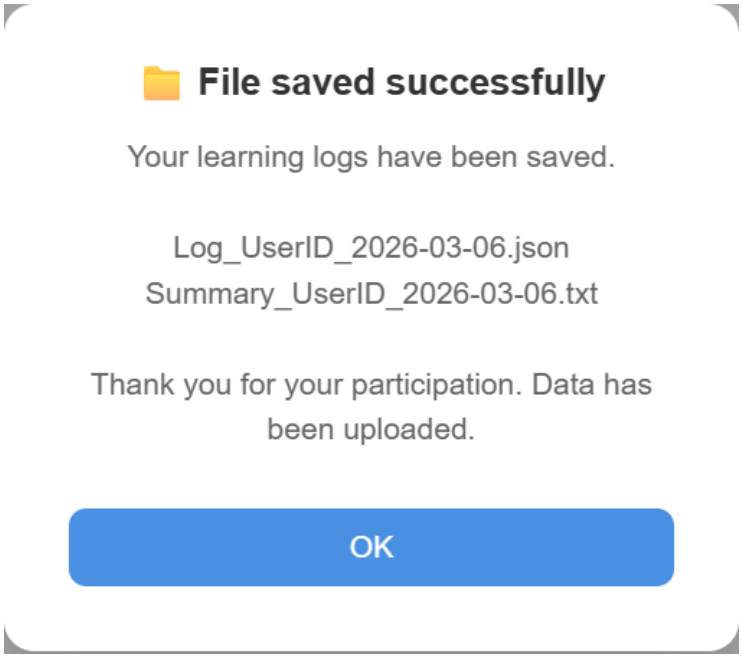

Figure 12. Dialog window for log export.

4.5. Requirement to Module Mapping

To keep the implementation aligned with the pedagogical goals, each requirement in Section 3 is mapped to concrete modules. Table 3 shows this mapping.

Table 3. Mapping between design requirements and implementation modules.

Design Requirement	Primary Module	Implementation Note
(1) Explainable logging	author-developed `logger.js` module + event API	captures ordered interaction records with timestamps and context
(2) Low-friction visual support	embedded visualizer panel	enables direct step-trace without leaving the practice page
(3) Non-replacement AI assistance	staged hint policy engine	controls L1/L2/L3 escalation and output style constraints
(4) Chapter-level reflection	chapter analytics + modal renderer	computes four-dimensional metrics and generates reflection text
(5) Classroom deployability	local Flask + local Ollama stack	supports privacy-preserving operation under limited connectivity

4.6. Prompt and Hint Policy Design

The hint generation pipeline uses a structured prompt template with four components: instructional role, task context, behavior context, and response constraint. The response constraint explicitly forbids direct final answers for standard hint requests. Level-specific policies are summarized below:

- **L1 policy:** Return conceptual explanation and one verification question.
- **L2 policy:** Return strategy decomposition and ordering advice.
- **L3 policy:** Return focused on mistakes guidance and minimal corrective clue.

When repeated L3 requests are detected within a short interval, the system adds a reflective message encouraging the learner to restate the problem intent and the expected intermediate state before requesting another hint. This mechanism is intended to reduce dependency loops.

4.7. Service Interfaces and State Transitions

To support maintainability and reproducibility, the backend services of the system are organized as explicit state transitions rather than implicit UI-driven side effects. Each learner session moves through a finite sequence of states: *Idle*, *TaskLoaded*, *Editing*, *Executed*, *HintRequested*, and *ChapterReviewed*. Transitions are triggered by validated API events organized into five core services:

- **Task service (*Idle* → *TaskLoaded*):** Loads the chapter and problem context, and initializes session metadata.
- **Execution service (*Editing* → *Executed*):** Runs the learner's code and stores attempt-level outcomes.
- **Hint service (*Executed* → *HintRequested*):** Generates staged assistance with context constraints.
- **Reflection service (→ *ChapterReviewed*):** Computes chapter indicators and returns the reflection package.
- **Export service (Any state → *Idle*):** Persists and exports log artifacts for review.

This explicit state model helps avoid inconsistent logs and duplicate events in high-frequency interactions. It also enables clearer debugging and auditing, since each event can be validated against an expected source state.

In practical terms, state-aware handling is especially useful when students rapidly alternate between running code and requesting hints. Without state guards, systems may

produce out-of-order logs or stale feedback contexts. The *PyPLAS* backend prevents this by attaching transition checks and timestamps at service boundaries.

4.8. Data Lifecycle and Privacy Handling

PyPLAS stores behavior data in an append-only local log format during active sessions. Aggregation results are generated on demand for chapter-based reflection, and only the derived indicators are displayed to students unless full log export is explicitly requested.

The data lifecycle has four stages: capture, aggregate, reflect, and export. During *capture*, raw events are written with minimal personally identifiable information; user names are replaced by local IDs. During *aggregation*, events are transformed into features at the chapter level. During *reflection*, only necessary indicators are sent to the AI service. During *export*, learners or teachers can obtain *JSON* and text reports for external analysis.

This lifecycle design supports two principles: *data minimization* and *educational transparency*. Students can see what kind of actions are measured and how those actions influence reflection outputs.

4.9. System Availability

The *PyPLAS* platform has been introduced and used in several previous studies as a *Python*-oriented extension of *PLAS* for programming self-study [24]. These previous studies mainly focused on structured exercise problems and automatic answer correctness checking. In contrast, the present study extends the existing *PyPLAS* framework by adding a novice-friendly answer interface with integrated code behavior visualization, staged AI-assisted hints, and learning-log-based analytics. Therefore, the contribution of this work is not the initial development of *PyPLAS* itself, but the design and pilot evaluation of the extended support functions for high school *Python* learners. The current extended prototype has not been publicly released, and no open source license has been assigned at this stage.

4.10. Deployment and Operational Scenarios

The system supports three practical deployment scenarios:

- **Lab mode:** Pre-configured classroom PCs with shared baseline model settings.
- **Teacher laptop mode:** Single-machine demonstration with live hinting and reflection demo.
- **Student self-study mode:** Local installation for after-class practice and log export.

Regardless of the adopted deployment mode, conducting chapter reflection at natural instructional checkpoints, such as the end of a weekly unit, is recommended when a full chapter has been completed, rather than immediately after completing only a small number of problems. Because this approach allows AI-generated feedback to use a larger amount of behavioral data, it can identify more coherent learning patterns and reduce interpretive bias caused by isolated or accidental errors in a single task.

4.11. Implementation Risks and Mitigation

Implementation-level risks include the model latency on a lower-end hardware, excessive hint requests in difficult chapters, and inconsistent logs caused by abrupt browser closure. To address these risks, *PyPLAS* includes the selection of the deployment-time model size (e.g., 1.5 B or 3 B according to available hardware), request throttling for repeated hints, and periodic log flush operations.

In addition, the UI includes explicit status feedback for AI response progress to reduce confusion during inference delays. If an AI service is unavailable, the system gracefully degrades to a mode without AI support and preserves the core functionality of the practice.

5. Evaluation

In this section, we show the evaluation of the proposal.

5.1. Evaluation Scope

This pilot evaluation provides initial evidence of practical value regarding the implemented interface for *PyPLAS* in high school programming classes. As this paper focuses on classroom feasibility and preliminary usability evaluation, we report descriptive behavioral trends, questionnaire responses, and instructional observations rather than establishing causal inference. The evaluation was conducted according to the research questions stated in Section 1. Specifically, RQ1 focuses on embedded visualization and learners' perceived understanding of runtime behavior, RQ2 focuses on progressive AI hints and attempt/hint-use patterns, RQ3 focuses on four-dimensional learning analytics and teacher interpretation, and RQ4 focuses on the usability and acceptance of the integrated interface. Accordingly, the numerical results are used to describe the observed tendencies in this pilot setting. They are not intended to support statistical generalization or causal claims.

5.2. Application Setup

To examine the preliminary feasibility and usability of the proposed *PyPLAS* system, we conducted a pilot application with a group of high school students. The participants were 20 high school students with no prior programming experience.

For this application, we prepared an introductory *Python* assignment consisting of six fundamental problems. The problem set was designed to cover basic syntax, conditional branches, and loops. The students were required to solve these problems sequentially using the *PyPLAS* interface. During problem solving, they had the full access to the embedded *Python Tutor* visualization and the progressive AI hints. For operation in a classroom, the *PyPLAS* web service and local AI components were launched through *Docker* containers [25] on ordinary PCs. After completing the assignment, they were asked to view their reflection feedback at the chapter level and answer a subjective questionnaire.

The questionnaire was designed as an ad hoc instrument for this pilot application. Its questions were made to correspond to the following four research questions: Q1 examined learners' perceived understanding of runtime behavior, Q2 examined their perception of progressive AI hints, Q3 examined the usefulness of four-dimensional reflection feedback, and Q4–Q5 examined interface usability and frustration reduction. They were reviewed by the authors and collaborating teachers to check their relevance to the learning activities and evaluation focus. However, the questionnaire was applied to a small pilot sample and contained only five questions addressing different aspects of the system. Formal construct validation and reliability analysis will be conducted in future work. Therefore, the questionnaire results are interpreted only as descriptive indicators of the students' subjective perceptions, not as validated measurements of learning effectiveness.

5.3. Application Results

The solving results for each problem instance are summarized in Table 4. The table includes the target grammar concepts, the number of students who successfully completed each problem, the average number of answer submissions (attempts), the average number of AI hint requests, and the average solving time per student.

As shown in Table 4, the completion rate was high, with 19 of the 20 students finishing all assignments. For the basic grammar problems (P1 and P2), the average number of attempts remained below 2.0, and solving time was short. However, as the logical complexity increased in the loop-related problems (P4, P5, and P6), attempts, hint requests, and solving time also increased. This trend reflects the increasing conceptual difficulty of loop-related

tasks. The longest average solving time was observed in P5 (while loop), where students spent an average of 14.5 min. Notably, these metrics decreased slightly in P6 as students became more familiar with loop structures. Despite the difficulty of P5 and P6, the average number of attempts remained below 4.5, suggesting preliminary behavioral evidence that integrated visualization and progressive hints may help limit excessive trial-and-error behavior in this pilot context.

Table 4. Problem-solving results in the pilot application ($N = 20$).

ID	Target Grammar	Completed	Avg. Attempts	Avg. Hints	Avg. Time (min)
P1	Output and variables	20	1.4	0.1	2.5
P2	Arithmetic operations	20	1.9	0.4	4.2
P3	Conditional branch (if–else)	20	2.5	1.2	7.1
P4	For-loop basics	20	3.2	1.8	9.8
P5	While-loop with condition	19	4.4	2.7	14.5
P6	List iteration (loop + condition)	19	3.8	2.3	12.0

5.4. Questionnaire Results

To investigate the subjective perception, the students answered a questionnaire consisting of five items. Each item was rated on a 5-point Likert scale, ranging from 1 (strongly disagree) to 5 (strongly agree). The results are summarized in Table 5.

Table 5. Results of the subjective questionnaire (5-point Likert scale).

ID	Question Item	Mean ± SD
Q1	The embedded Python Tutor helped me understand how variables change during execution.	4.45 ± 0.60
Q2	The L1/L2/L3 progressive hints guided my thinking instead of just giving me the answer.	4.15 ± 0.81
Q3	I found the four-dimensional reflection feedback useful for understanding my learning habits.	3.80 ± 0.89
Q4	The system interface was intuitive and easy to use without jumping to other websites.	4.30 ± 0.73
Q5	Overall, this system reduced my frustration when encountering programming errors.	4.25 ± 0.71

Descriptively, the relatively high mean scores for Q1 (4.45) and Q4 (4.30) are related to RQ1 and RQ4, suggesting that students perceived the embedded visualization as useful for understanding runtime behavior and accepted the integrated interface positively in this pilot setting. The descriptive result for Q2 (4.15) suggests that students perceived the progressive hint policy as guidance for thinking rather than as direct answer provision. Although Q3 was positive (3.80), its relatively lower mean and larger standard deviation (0.89) accurately reflect a common classroom reality: while some students benefit from advanced analytics, some high school students may still require teacher explanations to fully interpret the reflection feedback.

5.5. Discussion

The pilot results should be interpreted in relation to the pedagogical rationale discussed in Section 2. First, the positive response to embedded visualization suggests that making runtime behavior visible within the same problem-solving interface may support learners' active inspection of variable changes and control flow. This finding is consistent with the design intention of supporting mental model construction in introductory programming practice. However, because this study did not include a control group,

the result should be interpreted as preliminary perceptual evidence rather than proof on learning improvement.

Second, the attempt and hint-use patterns suggest that progressive AI hints may help learners continue problem solving without immediately receiving final code answers. The L1/L2/L3 hint policy functions as staged guidance in the pilot context, but its instructional effect requires further validation through controlled or quasi-experimental studies. Therefore, the present result indicates the practical feasibility of non-replacement AI assistance rather than a confirmed causal effect on programming performance.

Third, the four-dimensional analytics provided interpretable process information for teachers. During this application, teachers observed several illustrative learner profiles from the combination of engagement, reasoning strategy, learning pace, and tool usage. First, *active reasoning learners* steadily used visualization to trace variables before requesting hints (R_{val} high and $CV_{\text{val}} < 1.0$). Second, *trial-and-error learners* relied on rapid, unstable submission intervals without effectively using the visualizer (R_{val} low and $CV_{\text{val}} > 1.2$). Third, *support-seeking learners* requested hints relatively frequently but also used visualization, suggesting that they were actively seeking external support while still engaging with program behavior. Fourth, *low-engagement learners* showed very short active time, incomplete task coverage, and limited use of both visualization and hints, suggesting that teachers may need to check their task comprehensions, motivations, or operational difficulties carefully before interpreting programming ability. These profiles may help teachers consider targeted strategy coaching for individual students, while avoiding treating the analytics as an automatic judgment of learner ability. This observation addresses RQ3 and provides preliminary support for the potential practical value of *PyPLAS* analytics in pedagogical interpretation.

Overall, the contribution of this study is not limited to implementing a technical system. The proposed interface operationalizes a process-aware learning support workflow for novice programming education by connecting runtime observation, staged non-replacement feedback, event-stream logging, and chapter-level reflection. This workflow provides a basis for supporting learners during problem solving and for helping teachers interpret learning behaviors. At the same time, the evidence remains preliminary because the evaluation was descriptive and exploratory. Due to the lack of inferential analyses with the small-scale pilot design, these observations should be regarded as an exploratory foundation rather than as a confirmed causal evidence of learning improvement.

5.6. Limitations

First, although the proposed interface was designed with reference to active learning, mental model construction, formative feedback, self-regulated learning, and learning analytics, the theoretical basis of the four-dimensional learning analysis model remains limited. In particular, the current dimensions and thresholds were selected heuristically for classroom interpretability and have not yet been fully validated against established pedagogical or psychometric frameworks.

Second, this application involved a single cohort ($N = 20$) in one high school, which limits the generalizability of the results across different countries, schools, curricula, learner populations, and device conditions. Therefore, the present findings should be interpreted as results from a small-scale pilot application rather than as broadly generalizable evidence.

Third, the questionnaire was designed as an ad hoc instrument for this pilot study. Although the items were aligned with the research questions and reviewed for relevance by the authors and collaborating teachers, formal construct validation and reliability analysis, such as Cronbach's alpha, were not conducted due to the small sample size and the limited number of questions. Therefore, the questionnaire results should be interpreted as descrip-

tive indicators of students' subjective perceptions rather than validated measurements of learning effectiveness.

Finally, this study did not include inferential analyses. Because the pilot application did not include a control group and was not designed as a controlled experiment, we reported descriptive behavioral trends, questionnaire responses, and teacher observations rather than inferential statistical tests or significant contrasts. Consequently, the observed tendencies and profile interpretation thresholds remain exploratory and require recalibration and validation using larger datasets in future controlled or quasi-experimental studies.

6. Conclusions

This paper presented a novice-friendly *PyPLAS* interface that integrates code behavior visualization, event stream learning logs, and AI-assisted feedback into a unified learning workflow for *Python* education at the high school level. To address the inherent difficulties that high school students may face when transitioning from natural language to structured programming logic, the proposed system connects the problem solving, runtime observation, adaptive hints, and reflection at the chapter level into a continuous, self-regulated cycle. Rather than treating coding practice and post-task reflection as isolated activities, *PyPLAS* is intended to help students observe and reason about dynamic program execution while receiving structured, phase-appropriate guidance.

This study contributes to introductory programming education across interface design, learning analytics, and practical deployment. Specifically, it makes three practical contributions. First, it integrates *execution visualization* and *staged AI hints* directly into the answer interface, helping novice learners inspect program behavior and receive guidance without switching to external tools. Second, it provides *process-aware learning analytics* based on event-stream logs and four interpretable dimensions, enabling teachers to understand learner engagement, reasoning strategy, learning pace, and tool usage. Third, it presents a *classroom-oriented local deployment* approach using *Docker*, *Flask*, and *Ollama*, with the aim of supporting privacy-preserving operation under ordinary school infrastructure.

The pilot application with high school students provided preliminary observations suggesting the potential practical value of the proposal. Objective interaction logs suggested that the progressive L1/L2/L3 hint policy could function as staged guidance rather than as immediate answer provision. Concurrently, subjective questionnaire responses indicated that students perceived the embedded stepwise visualization as useful for understanding variable state transitions and accepted the integrated interface positively. Furthermore, teacher observations suggested that the four-dimensional analytics profiles may be practically interpretable for instructional reflection. By reviewing indicators such as the coefficient of variation of submission intervals and visualization dependency ratios, teachers could preliminarily distinguish between active reasoning and trial-and-error patterns.

Educationally, *PyPLAS* shifts the focus from correctness-only assessment to process-aware learning support. In conventional programming environments for high school students, students often receive only binary outcomes (correct/incorrect) or cryptic compiler errors. This limited feedback makes it difficult for both learners and teachers to identify exactly why a student's mental model is flawed. By capturing how learners interact with debugging aids, how rapidly they resubmit flawed code, and when they escalate hint requests, *PyPLAS* makes the learning process more visible. This process-oriented information may help teachers consider proactive interventions, such as encouraging visualization-first diagnosis for learners who repeatedly submit unstable code without tracing execution states. For learners, this workflow is intended to support reflection on their own learning habits without encouraging excessive dependency on direct AI-generated solutions.

Given the small-scale pilot design, the ad hoc nature of the questionnaire, and the lack of inferential analyses discussed in Section 5.6, the present findings in this paper should be interpreted as an exploratory and preliminary evidence rather than as a definitive proof of instructional effectiveness. Accordingly, this study primarily demonstrates the technical feasibility and practical usability of the proposed interface in a pilot classroom context.

Future work will focus on three main directions to build upon the initial findings. First, we will expand the system's deployment to multiple classes and schools, collecting longitudinal, multi-week data to statistically validate and recalibrate the behavioral thresholds, thereby improving the stability and generalizability of the learner profiles. Second, we aim to ground the four-dimensional analytics model in stronger pedagogical and learning-analytics frameworks, while refining the prompt engineering templates for chapter-level feedback generation so that the AI-generated explanations better match the proficiency levels and learning phases of individual students. Third, we will develop comprehensive, teacher-facing analytics dashboards to facilitate cross-chapter progression tracking, and we will conduct larger quasi-experimental studies with control groups to more rigorously examine the instructional impact and long-term learning outcomes of the *PyPLAS* framework.

Author Contributions: Conceptualization, Z.F. and N.F.; methodology, Z.F. and Z.Z.; software, Z.F., Z.Z. and Y.Z.; validation, Z.F., W.-C.K., Y.-F.L. and P.-K.T.; formal analysis, Z.F.; investigation, Z.F., W.-C.K., Y.-F.L. and P.-K.T.; resources, N.F. and Y.Z.; data curation, Z.F.; writing—original draft preparation, Z.F.; writing—review and editing, N.F., Z.Z., Y.Z., W.-C.K., Y.-F.L. and P.-K.T.; visualization, Z.F.; supervision, N.F.; project administration, N.F. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: This study was based on the secondary analysis of pre-existing, fully anonymized educational activity data, and no personally identifiable information was accessible to the researchers. As supporting basis, under Japan's Act on the Protection of Personal Information, "anonymously processed information" means information processed so as neither "to be able to identify a specific individual" nor "to be able to restore the personal information" https://www.ppc.go.jp/files/pdf/APPI_english.pdf (accessed on 17 May 2026). In addition, the Japanese Ethical Guidelines for Medical and Biological Research Involving Human Subjects state that the Guidelines do not apply to research utilizing only: "Existing information that is not information relating to an individual"; and "Anonymized personal information that has already been created" https://www.mhlw.go.jp/content/001457376.pdf (accessed on 17 May 2026).

Informed Consent Statement: Prior to the implementation of the study, informed consent for participation was obtained by the collaborating school from all subjects (high school students), as well as their guardians or teachers.

Data Availability Statement: The behavioral log schema and anonymized sample records used in this study are available from the corresponding author upon reasonable request, due to privacy and ethical restrictions regarding student behavioral data.

Acknowledgments: The authors thank collaborating teachers and students for their feedback during the pilot use of the system.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

AI	Artificial Intelligence
LLM	Large Language Model
PLAS	Programming Learning Assistant System
PyPLAS	Python Programming Learning Assistant System

References

1. Top Programming Languages. 2025. Available online: https://spectrum.ieee.org/top-programming-languages-2025 (accessed on 17 May 2026).
2. Python Software Foundation. Python Official Website. Available online: https://www.python.org/ (accessed on 1 March 2026).
3. Aung, S.T.; Funabiki, N.; Lu, X.; Wai, K.H.; Jing, Y.; Kao, W.-C. An implementation of Java programming assistant system in university course. In Proceedings of the International Conference on Computers in Education, Taichung, Taiwan, 8–11 July 2024.
4. Lu, X.; Funabiki, N.; Aung, S.T.; Kyaw, H.H.S.; Ueda, K.; Kao, W.-C. A study of grammar-concept understanding problem in C programming learning assistant system. *ITE Trans. Media Technol. Appl.* **2022**, *10*, 198–207. [CrossRef]
5. Pallets. Flask Documentation. Available online: https://flask.palletsprojects.com/ (accessed on 1 March 2026).
6. Ollama. Available online: https://ollama.com/ (accessed on 1 March 2026).
7. Guo, P.J. Online Python Tutor: Embeddable web-based program visualization for CS education. In Proceedings of the 44th ACM Technical Symposium on Computer Science Education, Denver, CO, USA, 6–9 March 2013; p. 579.
8. van der Werf, V.; Aivaloglou, E.; Hermans, F.; Specht, M. What does this Python code do? An exploratory analysis of novice students' code explanations. In Proceedings of the 10th Computer Science Education Research Conference, Virtual, 22–23 November 2021; pp. 94–107.
9. Feeley, M.; Melancon, O. Teaching programming to novices using the codeBoot online environment. *arXiv* **2022**, arXiv:2207.12702. [CrossRef]
10. Paiva, J.C.; Queirós, R.; Leal, J.P.; Swacha, J.; Miernik, F. Managing gamified programming courses with the FGPE platform. *Information* **2022**, *13*, 45. [CrossRef]
11. zyBooks (formerly Coding Rooms). Available online: https://www.codingrooms.com/ (accessed on 17 May 2026).
12. Suriyaarachchi, H.; Denny, P.; Nanayakkara, S. Investigating the use of productive failure as a design paradigm for learning introductory Python programming. In Proceedings of the ACM Technical Symposium on Computer Science Education, Pittsburgh, PA, USA, 26 February–1 March 2025.
13. Siemens, G.; Long, P. Penetrating the fog: Analytics in learning and education. *EDUCAUSE Rev.* **2011**, *46*, 30–40.
14. Bogarín, A.; Cerezo, R.; Romero, C. A survey on educational process mining. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* **2018**, *8*, e1230. [CrossRef]
15. Fowler, M. Event Sourcing. Available online: https://martinfowler.com/eaaDev/EventSourcing.html (accessed on 1 March 2026).
16. Khanmigo. AI-Powered Tutor and Teaching Assistant. Available online: https://www.khanmigo.ai/ (accessed on 1 March 2026).
17. GitHub Copilot. Your AI Pair Programmer. Available online: https://github.com/features/copilot (accessed on 1 March 2026).
18. Gasiba, T.E.; Iosif, A.-C.; Kessba, I.; Amburi, S.; Lechner, U.; Pinto-Albuquerque, M. May the source be with you: On ChatGPT, cybersecurity, and secure coding. *Information* **2024**, *15*, 572. [CrossRef]
19. Fan, G.; Liu, D.; Zhang, R.; Pan, L. The impact of AI-assisted pair programming on student motivation, programming anxiety, collaborative learning, and programming performance: A comparative study with traditional pair programming and individual approaches. *Int. J. STEM Educ.* **2025**, *12*, 16. [CrossRef]
20. Akçapınar, G.; Sidan, E. AI chatbots in programming education: Guiding success or encouraging plagiarism. *Discov. Artif. Intell.* **2024**, *4*, 49. [CrossRef]
21. Ifenthaler, D.; Schumacher, C. Student perceptions of privacy principles for learning analytics. *Educ. Technol. Res. Dev.* **2016**, *64*, 923–938. [CrossRef]
22. Shi, N. Improving undergraduate novice programmer comprehension through case-based teaching with roles of variables to provide scaffolding. *Information* **2021**, *12*, 424. [CrossRef]
23. Hui, B.; Yang, J.; Cui, Z.; Yang, J.; Liu, D.; Zhang, L.; Liu, T.; Zhang, J.; Yu, B.; Lu, K.; et al. Qwen2.5-Coder Technical Report. *arXiv* **2024**, arXiv:2409.12186. [CrossRef]

24. Hay Mar Shwe, S.H.M.; Funabiki, N.; Syaifudin, Y.W.; Htet, E.E.; Kyaw, H.H.S.; Tar, P.P.; Win Min, N.; Myint, T.; Thant, H.A.; Kao, W.-C. Value trace problems with assisting references for Python programming self-study. *Int. J. Web Inf. Syst.* **2021**, *17*, 287–299. [CrossRef]
25. Docker. Docker Official Website. Available online: https://www.docker.com/ (accessed on 1 March 2026).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.