CPIAE: A Non-Blocking Edge-Based Python Architecture for AI-Assisted Programming Education with Context-Aware Feedback

Guillermo Sentoni[1]

[1] Universidad Nacional de La Matanza, F. Varela 1903, Bs. As. Argentina
gsentoni@gmail.com

Abstract. This paper presents CPIAE, an edge-based architecture that delivers scientific Python environments as self-contained web artifacts for programming education. The system runs a complete Python runtime in the browser via Pyodide/WebAssembly, with no server-side infrastructure. The core contribution is a non-blocking execution model built on a dedicated Web Worker, which decouples computation from the UI thread and preserves interactivity during intensive tasks such as neural network training. This resolves the interface-freezing limitation that affects all existing synchronous browser-based Python environments. CPIAE also integrates a context-aware Gemini 2.5 Flash AI assistance module that automatically builds large language model prompts from the student's live code and runtime error output. A pedagogical 3D avatar with phoneme-synchronized speech synthesis narrates the AI responses. Three additional technical contributions round out the system: a decoupled module architecture with a shared `EventTarget` event bus; an import cache eliminating redundant library downloads; and a prompt engineering strategy grounding feedback in the live execution state. The system was deployed across a complete university neural computing course in Moodle. A pilot study with 15 undergraduate students showed that around 80% of runtime errors were resolved autonomously without instructor intervention. The single-artifact model was also verified on a smartphone, confirming viability on any standards-compliant browser. These results provide initial empirical evidence of the technical robustness and operational viability of edge-based execution combined with situated AI feedback.

Keywords: edge computing, WebAssembly, Web Worker, AI assistance, Pyodide.

CPIAE: Arquitectura Python No Bloqueante Basada en Edge Computing para la Educación en Programación Asistida por IA con Retroalimentación Contextual

Resumen. Este trabajo presenta CPIAE, una arquitectura edge que provee entornos Python científicos como artefactos web autocontenidos para la educación en programación. El sistema ejecuta un runtime Python completo en

el navegador mediante Pyodide/WebAssembly, sin infraestructura de servidor. La contribución central es un modelo de ejecución no bloqueante basado en un Web Worker dedicado, que desacopla la computación del hilo de interfaz y preserva la interactividad durante tareas intensivas como el entrenamiento de redes neuronales. Esto resuelve el congelamiento de interfaz que afecta a todos los entornos Python síncronos en el navegador. CPIAE integra además un módulo de asistencia IA Gemini 2.5 Flash que construye automáticamente prompts de modelo de lenguaje de gran escala a partir del código activo y la salida de error del estudiante. Un avatar pedagógico 3D con síntesis de voz sincronizada a fonemas narra las respuestas. Tres contribuciones técnicas adicionales completan el sistema: una arquitectura modular desacoplada con bus EventTarget compartido; una caché de imports que elimina descargas redundantes; y una estrategia de ingeniería de prompts anclada al estado de ejecución en vivo. El sistema se desplegó en un curso universitario completo de computación neuronal en Moodle. Un estudio piloto con 15 estudiantes de grado mostró que alrededor del 80 % de los errores de ejecución se resolvieron autónomamente sin intervención docente. El artefacto de archivo único fue además verificado en un teléfono inteligente, confirmando viabilidad en cualquier navegador compatible con estándares. Estos resultados aportan evidencia empírica inicial sobre la solidez técnica y viabilidad operativa de la ejecución edge combinada con retroalimentación IA situada.

Palabras clave: edge computing, WebAssembly, Web Worker, asistencia IA, Pyodide.

1 Introduction

The provisioning of interactive execution environments for teaching programming-intensive disciplines poses recurring software engineering challenges in terms of scalability, reproducibility, security, and operational costs. The dominant approach — centralized notebook servers such as JupyterHub or Google Colab — depends on shared server infrastructure, introducing resource contention, complex isolation requirements, variable latency, and access friction that become structurally problematic in large-cohort educational settings (Pimentel et al., 2019). This paper addresses that gap in the context of a Moodle-based neural computing course, where infrastructure constraints and student heterogeneity make server-based platforms particularly costly.

CPIAE sits at the intersection of Software Engineering, browser-based execution architectures, and AI-assisted learning environments. It addresses a concrete operational need: a reproducible, infrastructure-free Python environment for a Moodle-based neural computing course with heterogeneous hardware cohorts. Executing Scikit-learn training loops — iterative, CPU-intensive — under in-browser WebAssembly with a responsive UI directly motivated CPIAE's Web Worker architecture. The zero-installation model targets contexts where environment setup is a barrier, not a learning objective: heterogeneous hardware cohorts, mobile-only students, and large-enrollment LMS (Learning Management System) deployments. For programs where environment configuration is itself a professional competency, server-based alternatives remain appropriate. Fig. 1 contrasts the traditional client–server model with the CPIAE server-edge model.

Neural computing education amplifies these difficulties. Training modest neural

network models requires iterative computation over multiple epochs — sustained load that scales linearly with cohort size in centralized architectures. Google Colab mitigates compute constraints via cloud GPUs, but imposes account requirements, session limits, and connectivity dependencies incompatible with LMS deployments. JupyterLite eliminates server-side execution via Pyodide, but still runs Python synchronously in the main browser thread, blocking the UI under load (Corlay, 2025).

Edge computing addresses this class of problems by shifting execution to the client (Satyanarayanan et al., 2015; Shi et al., 2016). CPIAE realizes this paradigm through Pyodide, which compiles CPython to WebAssembly via Emscripten (Droettboom, 2018; Jangda et al., 2019), enabling full in-browser execution of the scientific Python stack. CPython's WebAssembly target reached Tier 3 support in Python 3.14, consolidating Pyodide as the canonical approach for full-fidelity in-browser Python (Langa, 2025).

CPIAE makes three contributions beyond edge execution: (i) a decoupled module architecture with a shared `EventTarget` event bus, separating execution, AI assistance, and presentation; (ii) an import cache eliminating redundant library downloads, combined with an AI module that automatically embeds live code and runtime errors in each large language model (LLM) prompt; and (iii) a prompt engineering strategy grounding feedback in the live execution state, validated through a complete university course deployment.

2 Related Work and Positioning

2.1 Browser-based Python execution environments

Pyodide (Droettboom, 2018) is the foundational WebAssembly Python runtime used by all major browser-based platforms. JupyterLite builds on Pyodide to offer a serverless Jupyter distribution (Corlay, 2025), deployed at scale to serve hundreds of thousands of students. PyScript provides a higher-level HTML-embedded interface to the same runtime. All three execute Python in the main browser thread, relying on service workers or experimental shared memory for asynchronous operations — resulting in UI freezes under computationally intensive workloads.

Fig. 1. Traditional client–server execution model versus the CPIAE server-edge model. In CPIAE, all Python execution occurs locally in the student's browser.

2.2 AI-assisted programming education

Automated feedback for programming exercises has been a long-standing focus, spanning static analysis, test-based grading, and natural language generation (Keuning et al., 2018). Generative AI has expanded this scope, enabling natural-language explanations at scale (Lee & Moore, 2024). CREF (Yang et al., 2024) achieves up to 76.6% repair correctness with GPT-4 via tutor guidance and failing test cases. This figure and CPIAE's 80% autonomous error-resolution rate are not directly comparable: CREF measures syntactic correctness on predefined bugs under tutor guidance; CPIAE's figure reflects student-initiated resolution in an open-ended deployment. Both point to the same operational range for LLM-assisted error resolution. HypoCompass (Ma et al., 2024) takes a different angle: it trains students to debug LLM-generated code through a learning-by-teaching interface. Context sensitivity has emerged as the primary factor in feedback quality — fine-tuned LLMs producing context-sensitive error messages significantly improve novice problem-solving (Taylor et al., 2024). Scholz et al. (2025) further show that pedagogical feedback systems grounded in the student's actual submission context yield better outcomes. Parallel findings in data science education confirm measurable learning gains when AI feedback is anchored to observable task performance (Letteri & Vittorini, 2025).

The critical distinction of CPIAE's approach is automatic prompt construction from the live execution environment. Static LLM-based repair tools (such as those described above) operate on code submitted to a server-side system. CPIAE, by contrast, captures the student's active code and the exact runtime error at the moment of failure, then constructs the LLM prompt without requiring the student to copy, describe, or submit anything. Seeking help is reduced to a single button press.

Chen et al. (2024) show through SELF-DEBUGGING that incorporating code execution results substantially improves LLM repair performance — the theoretical basis for CPIAE's context injection. CPIAE operationalizes this insight in a zero-infrastructure educational setting.

2.3 Pedagogical avatars in computing education

Fink et al. (2024) document that AI-based avatars change the way students learn, with benefits depending on cognitive load conditions and task complexity; speech-synthesis agents in particular have shown value for engagement and retention in instructional contexts. Liu et al. (2026) describe an LLM-empowered arguing pedagogical agent with personalized avatar, movements, and voice. The CPIAE avatar (`TalkingBook3D`) adopts a minimalist approach consistent with findings on cognitive load: a stylized 3D book with phoneme-synchronized mouth animation and Web Speech API synthesis narrates AI responses, providing the auditory modality without the visual complexity of a humanoid avatar. This design deliberately reduces the cognitive overhead associated with over-realistic virtual agents (Drobnjak & Boticki, 2025).

2.4 Neural computing education infrastructure

Teaching neural computing at university level conventionally requires either local installation (Python, NumPy, Scikit-learn, Matplotlib) or server-based platforms (Google Colab, university Jupyter servers). Local installation introduces environment

inconsistency and configuration failures that consume instructional time; server-based platforms bring the scalability limitations described above. CPIAE fills this gap with a browser-executable environment covering the full Scikit-learn neural network API and Matplotlib visualization — validated through a proof-of-concept pilot deployment in an actual university course.

3 Requirements for Browser-Based Neural Computing

CPIAE was designed around a set of functional and non-functional requirements. The system provides a complete scientific Python stack: Pyodide loads NumPy, Matplotlib, Pandas, and Scikit-learn dynamically from CDN, with figures serialized to SVG strings and injected directly into the DOM. Contextual AI assistance relies on automatic prompt construction embedding the student's live code, `stdout/stderr` output, and full conversation history before each API call. Code persistence is handled through explicit LocalStorage save/load under user control. The entire environment is packaged as a single HTML artifact, with the Web Worker source embedded via a Blob URL to preserve the zero-dependency distribution model. On the non-functional side, the edge execution model achieves zero marginal scaling cost: each instance runs independently with no shared server resources. The dedicated Web Worker keeps the UI thread free during computation: the editor remains editable, interactive controls stay active, and the avatar continues rendering while Python executes in the background. The WebAssembly sandbox provides strong execution isolation — no OS or filesystem access. Environmental reproducibility is guaranteed by pinning the Pyodide runtime version, so all students execute against an identical stack. After initial load, all runtime assets are cached and the Worker operates fully offline, with no installation, account, or setup required.

4 CPIAE Architecture

4.1 Overview and distribution model

CPIAE is distributed as a single self-contained HTML file. All JavaScript logic, CSS, the Worker source code, and the default Python example are embedded within this artifact. External dependencies (Pyodide, CodeMirror, Three.js, Marked.js) are loaded from CDN at first use and cached by the browser for subsequent offline operation. This distribution model shifts the provisioning problem from runtime infrastructure management to artifact version control: releasing a new version of the course environment consists of distributing a new HTML file.

All system logic is organized into five modules coordinated through `window.CPIAE`, which owns the shared state (`CPIAE.state`), the inter-module event bus (`CPIAE.events`, an `EventTarget`), and the editor abstraction layer (`CPIAE.editor`). This namespace pattern prevents global variable pollution and provides a lightweight coordination mechanism requiring no build system or module bundler. Fig. 2 shows the system running in the browser.

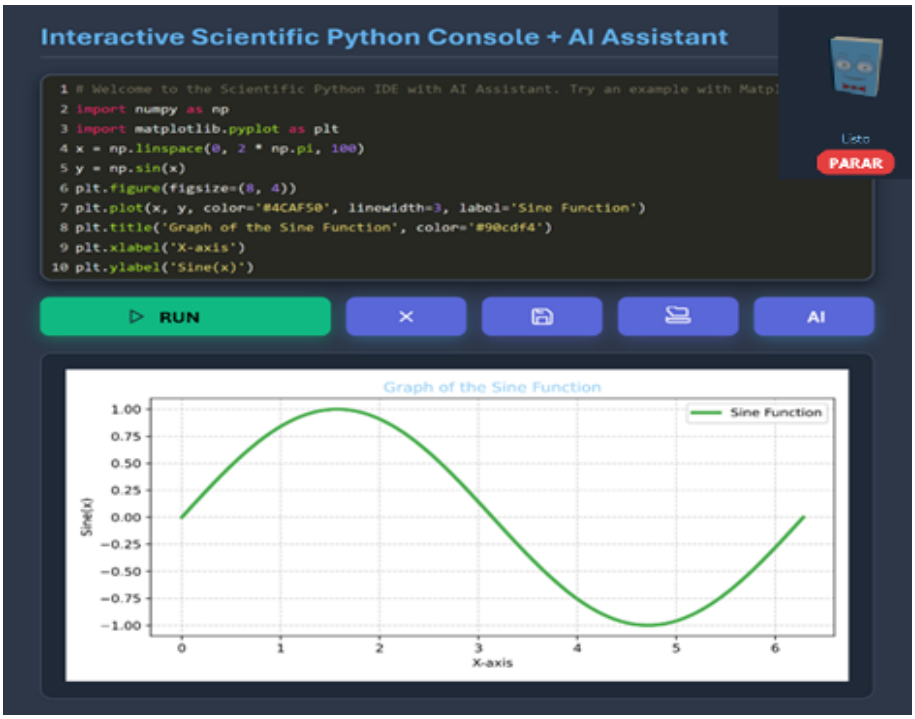

Fig. 2. CPIAE running in the browser: CodeMirror editor, execution control, and inline Matplotlib SVG output. The AI assistant and avatar activate via the AI button.

4.2 Module structure and inter-component communication

Bootstrap initializes the DOM, configures the CodeMirror 5 editor (Python mode, Monokai theme, line numbers), disables interactive controls until Pyodide signals readiness, and caches DOM references for subsequent access. Console manages the execution lifecycle: it receives user-initiated run events, sends code to the Worker via `postMessage`, renders `stdout/stderr` output to the output area, and injects Matplotlib SVG strings directly into the plot area DOM. When the Worker reports an error, Console does not call the Assistant directly — it dispatches a `console:error` CustomEvent on `CPIAE.events`. This indirection preserves module independence: the Assistant module subscribes to this event without Console knowing whether an assistant exists. Assistant builds contextual LLM prompts, communicates with Gemini 2.5 Flash, renders Markdown responses via Marked.js, and extracts suggested Python code via regex for one-click application. Avatar renders the Three.js scene and exposes only three public methods (init, speak, stop), isolating the 3D implementation from the rest of the system. `PyodideWorker` is the only module that runs off the main thread.

The `EventTarget` bus (`CPIAE.events`) is the primary inter-module communication mechanism. It enables future modules — for example, an exercise evaluation module or a session recorder — to subscribe to execution events without modifying existing modules. This open/closed design supports the extensibility requirements of a living educational tool.

4.3 The Web Worker: non-blocking Python execution

The central architectural contribution of CPIAE is the execution of Pyodide in a dedicated Web Worker. In all prior synchronous browser-based Python environments (including JupyterLite's kernel), Python execution occupies the main browser thread. For computationally lightweight code this is acceptable, but neural network training with Scikit-learn — even on small datasets — involves iterative gradient descent loops that can run for several seconds or minutes. Under the synchronous model, the browser UI freezes for the duration: no scroll, no button response, no visual feedback. This is educationally damaging because it removes the student's ability to cancel a runaway loop or interact with the interface while computation proceeds.

The Web Worker resolves this by executing Pyodide in a separate OS thread. The Worker is instantiated from source code embedded in the HTML artifact via the Blob URL pattern:

```
const src = document.getElementById('pyodide-worker').textContent;
const blob = new Blob([src], { type: 'text/javascript' });
const worker = new Worker(URL.createObjectURL(blob));
```

This approach preserves the single-artifact distribution model: the Worker source is not a separate file that must be co-located or served with specific MIME types. The Worker is instantiated once and reused across executions; re-instantiation on each run would require re-downloading the ~20 MB Pyodide runtime.

The message protocol between the main thread and the Worker is deliberately minimal, exposing only the surface area required for current functionality and leaving room for future extension without breaking changes.

4.4 Import cache and package management

Pyodide's `loadPackagesFromImports()` scans Python source code for import statements and downloads required packages from a CDN. This operation is expensive (hundreds of milliseconds for packages already in the CDN cache, seconds for cold loads) and unnecessary when the student reruns code that imports the same packages as the previous execution.

CPIAE implements a Worker-side import cache (a JavaScript Set, `cachedImports`) that tracks resolved packages across executions. Before each run, the function `extractTopLevelImports()` parses the code with regular expressions to extract statically declared imports. If all detected modules are already in the cache, `loadPackagesFromImports()` is skipped entirely. Dynamic import patterns (`__import__`, `importlib.import_module`) force a full scan as a safety measure. This optimization is particularly valuable in interactive neural computing exercises, where students frequently modify a training function and rerun without changing the import block.

4.5 Matplotlib SVG pipeline

A design decision with significant implications for rendering quality is the choice to serialize Matplotlib figures to SVG strings within the Worker rather than PNG bitmaps.

After executing user code, the Worker iterates over all active figure numbers, calls `plt.gcf()`.savefig(buf, format='svg'), and collects the SVG strings into a list that is transmitted via `postMessage`. The main thread receives these strings and injects them directly into the plot area DOM using innerHTML.

This approach produces resolution-independent vector graphics that scale correctly on high-DPI displays and zoom without degradation — important for neural network visualizations such as decision boundary plots and training loss curves. It also avoids the base64 encoding overhead associated with PNG transmission and the canvas element lifecycle management that PNG-based approaches require.

The `plt.show()` call is stripped from user code before execution (a simple replace() on the code string) because Pyodide's matplotlib backend does not support the interactive display loop; figures are captured and transmitted explicitly after execution.

4.6 Context-aware AI assistance

The AI assistance module activates in two modes: (1) manual, where the student types a question and presses "Ask"; and (2) automatic, triggered when the Worker reports a Python runtime error. In the automatic mode, the module subscribes to the `console:error` event dispatched by the Console module, enabling error-triggered assistance without coupling Console to Assistant.

In both modes, the `buildAssistantContext()` function constructs a prompt that includes: the student's current question (or a default error analysis request), the full current code from the editor, and the complete console output (stdout + stderr). This context is appended to the full conversation history (`CPIAE.state.chatHistory`) before submission to the Gemini API, enabling multi-turn interactions where the LLM can refer to previous exchanges.

The system prompt instructs Gemini to always respond in Spanish — reflecting the language of instruction of the host institution — use Markdown formatting, and wrap code suggestions in Python fenced code blocks. The latter convention enables reliable regex extraction of suggested code (pattern: `` ```python\s*([\s\S]*?)\s*``` ``) and conditional activation of the "Apply Code" button, which replaces the editor content with the suggestion on a single click.

To handle API rate limits in multi-user deployments, the module implements exponential backoff with up to five retries (delays of 1, 2, 4, 8, and 16 seconds) on HTTP 429 responses. RESOURCE_EXHAUSTED status terminates the retry loop immediately and surfaces a user-facing message.

4.7 Pedagogical avatar

The Three.js avatar (class `TalkingBook3D`) renders a 3D book with a canvas-textured face on the front cover. The face texture (class `FaceCanvas`) is redrawn each animation frame with updated eye, mouth, and blinking state. Mouth aperture is driven by the Web Speech API's onboundary event, which fires at word and sentence boundaries during speech synthesis. A heuristic function (`_heuristicOpenForChar`) maps the character at the current boundary index to an aperture target: vowels → 1.0, voiced consonants → 0.75, fricatives → 0.5, plosives → 0.45, silence/punctuation → 0.05. The aperture target is smoothed toward the current value each frame via linear

interpolation to produce natural-looking lip movement.

The avatar is architecturally isolated behind three public methods (init, speak, stop) on the `AvatarModule` object. The rest of the system never references Three.js or the Web Speech API directly. This design allows the avatar to be replaced, disabled, or upgraded without modifying any other module — relevant for deployments where TTS is unavailable or undesirable.

4.8 Security and execution isolation

Python code in CPIAE executes within the browser's WebAssembly sandbox. The Wasm memory model prohibits direct memory access outside the allocated region; the browser's sandbox prevents access to the operating system, local file system, and external network resources not explicitly enabled via CORS. This security model eliminates the primary risks of server-side shared-execution platforms: there are no persistent server processes, no shared state between student instances, and no possibility of one student's code affecting another's environment (Haas et al., 2017).

The practical consequence for institutional deployment is that CPIAE requires no security review of student-submitted code beyond what the browser itself enforces. This is a significant operational simplification relative to sandboxed server-side execution, which requires container isolation, resource quotas, and monitoring infrastructure.

5 Deployment in a Neural Computing Course

5.1 Course structure and CPIAE integration

CPIAE was deployed as the primary execution environment for a neural computing course at a public university, delivered through the Moodle LMS. The course covered the following topics over several weeks: perceptrons and the MLP (multilayer perceptron) architecture, activation functions and backpropagation, overfitting and regularization, Scikit-learn's MLPClassifier and MLPRegressor APIs, training curve analysis, classification and regression on standard datasets.

All practical exercises were distributed as CPIAE HTML artifacts embedded in Moodle pages via iframe, or as downloadable files accessible via QR code. Students executed every exercise — from simple NumPy array manipulations to multi-epoch MLP training on 2D classification datasets — directly in their browser without any local installation. The Scikit-learn MLPClassifier, with its iterative solver and configurable hidden layer sizes, constitutes the primary computational workload that motivated the Web Worker architecture: training runs of 200–500 epochs on datasets of 100–500 samples produce training times of 1–8 seconds in Pyodide/Wasm, during which the UI must remain responsive for the student to monitor progress indicators and cancel if needed.

5.2 Exercise design for browser-based neural computing

The course exercises were designed to exploit CPIAE's specific capabilities. Matplotlib inline SVG rendering was used extensively for visualizing decision boundaries after MLP training, plotting training loss curves epoch by epoch, and displaying confusion

matrices. The AI assistant was explicitly integrated into the exercise workflow: exercises included instructions to activate the assistant when encountering a specific type of error (e.g., convergence warnings, shape mismatches in input arrays), directing students to use the automatic error analysis feature rather than asking the instructor.

The single-artifact model enabled version-controlled exercise distribution: each week's exercises were a new HTML file with updated default code, distributed through Moodle's file repository. Students could save their solutions to LocalStorage and reload them in subsequent sessions.

5.3 Operational results

Throughout the course, no server-side execution infrastructure was required beyond the Moodle LMS itself (which served only static HTML files and course materials). Zero configuration support requests related to Python installation or environment setup were received — a marked contrast with the previous offering of the same course, which used a university Jupyter server and required an average of 45 minutes of setup support per student in the first two weeks.

The Web Worker architecture proved essential for the MLP training exercises: without it, browsers would have displayed "page unresponsive" warnings during 1–8 second training runs. With the Worker, the interface remained responsive throughout, allowing students to cancel long-running fits and modify hyperparameters without page reload. Interaction logs collected during the deployment showed that around 80% of runtime errors were resolved autonomously through the AI assistant without instructor intervention; this is an observational figure from the pilot, not a controlled experimental measurement. Taken together, these operational observations constitute the primary outcome of this proof-of-concept phase: they confirm that the non-blocking execution architecture functions correctly under authentic LMS conditions with real student workloads. Quantitative measurement of learning outcomes and comparative error-resolution rates are outside the scope of this PoC and are identified as directions for future controlled studies.

Although not an initial design requirement, the architecture was additionally tested on mobile hardware. Execution on a Samsung Galaxy S24 running mobile Chrome verified that the single-artifact model runs without modification on a smartphone. The complete pipeline—Pyodide initialization, package resolution, MLP training, and AI assistant interaction—executed successfully. This behavior is a direct architectural consequence of the zero-dependency distribution model and native WebAssembly support in mobile browsers, demonstrating that the edge execution model is viable on any device with a standards-compliant browser.

6 Conceptual Comparison with Existing Environments

The scalability advantage of the edge model is structural. In server-based architectures, service quality degrades as concurrent users compete for shared CPU, memory, and I/O resources; this degradation accelerates with concurrency (Satyanarayanan et al., 2015; Shi et al., 2016), as illustrated schematically in Fig. 3. In CPIAE, each student’s runtime executes independently in their own browser, so cohort size has no effect on individual execution quality. No capacity planning or infrastructure scaling is required.

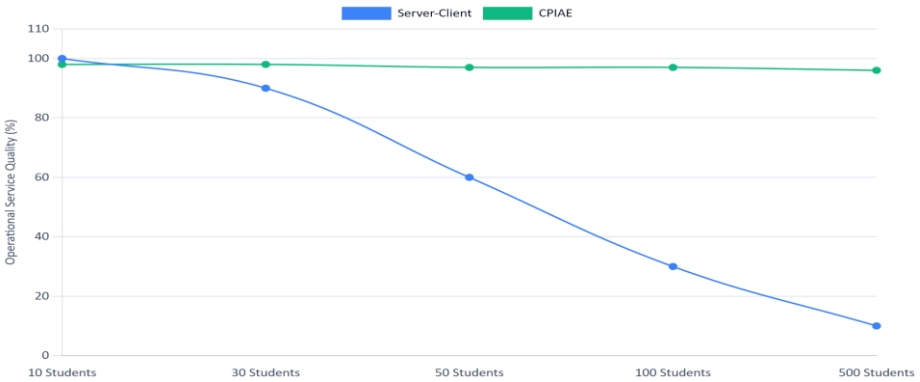

Fig. 3. Operational service quality as a function of concurrent users (schematic). Server-client quality degrades sharply beyond ~30 simultaneous sessions while CPIAE remains near-constant

Among the environments compared, CPIAE is the only one that combines zero server infrastructure, non-blocking execution under intensive computation, and integrated context-aware AI assistance — all validated in a complete university course. JupyterLite is the closest architectural relative (serverless, Pyodide-based) but runs Python on the main browser thread and provides no AI assistant. Google Colab avoids blocking through cloud resources, but requires a Google account and session time limits, and its AI assistance relies on manual prompting with no automatic execution-state context. JupyterHub demands server infrastructure with no AI assistance. Local environments such as VS Code add installation overhead, and their AI extensions do not capture runtime context automatically.

7 Discussion

7.1 The Web Worker as a required architectural primitive

The deployment confirmed a design hypothesis that was architectural rather than empirical: non-blocking execution is not a quality-of-life improvement but a functional requirement for computationally intensive educational Python. Under the previous synchronous model, MLP training runs exceeding 1–2 seconds triggered the browser's "page unresponsive" dialog, requiring a page reload that lost all execution state. The Web Worker eliminates this failure mode entirely. Any browser-based Python environment targeting scientific computing or machine learning education should treat the Worker architecture as a required feature, not an optimization.

7.2 Context injection as the primary differentiator of AI assistance quality

The primary differentiator in CPIAE is automatic construction of the prompt context from the live execution state. Capturing verbatim source code and runtime telemetry at the moment of failure eliminates the friction of manual error reporting — the student presses one button rather than composing a description. The observed pattern of autonomous error resolution suggests this friction reduction is the operative

mechanism, consistent with Yang et al. (2024). This design entails a recognized pedagogical trade-off. Automating prompt construction eliminates the cognitive step of reading, localizing, and articulating the error — a process associated with autonomous problem-solving development (Ma et al., 2024; Taylor et al., 2024). The one-click "Apply Code" mechanism extends this trade-off to the correction stage. These are limitations of the current design, not incidental omissions: future work should investigate scaffolded variants that progressively reduce automation as student competence develops, consistent with the learning-by-teaching approach of HypoCompass (Ma et al., 2024).

7.3 Deployment Considerations, Security, and Study Scope

The pedagogical avatar was received positively during the deployment, supporting affect regulation at the moment of failure; the module is architected as an elective scaffolding layer that can be disabled at the instructor's discretion without affecting any other system component.

Regarding study scope, this work constitutes a pilot study and proof-of-concept validation: its primary objective was to confirm that the non-blocking execution architecture performs correctly under authentic LMS conditions (Moodle), which it did. The 15-student deployment reflects this scoping. The 80% error-resolution rate reported in the abstract is an observational figure derived from interaction logs collected during the pilot — it documents what occurred during the deployment but was not obtained under controlled experimental conditions. Longitudinal learning outcomes and comparative error-resolution rates between CPIAE and alternative tools were not the target of this phase. Future controlled studies should compare CPIAE against a general-purpose chat interface lacking automatic context injection to isolate the pedagogical contribution of the situated feedback mechanism.

8 Conclusions

This paper presented CPIAE, a browser-based architecture for provisioning scientific Python environments as self-contained web artifacts, piloted through a proof-of-concept deployment in a complete university neural computing course. The architecture satisfies the core non-functional requirements identified for institutional-scale deployment — scalability by construction, strong WebAssembly execution isolation, environmental reproducibility, and zero-configuration access — through three technical contributions not found in existing browser-based Python environments.

The dedicated Web Worker architecture resolves the UI-blocking limitation that affects all synchronous browser-based Python runtimes, enabling the execution of computationally intensive operations (MLP training, iterative numerical methods) without degrading interface responsiveness. The operational deployment in the neural computing course confirmed that this is a functional requirement for scientific computing education, not merely an optimization. The automatic context injection mechanism — constructing LLM prompts from the student's live code and runtime error output — confirmed the practical value of grounding AI feedback in the observable execution state.

The single-artifact distribution model has real limits — no session persistence, no

collaboration — but in the LMS context it outperformed server-based alternatives on every operational dimension: zero environment-related support overhead, reproducible distribution via version-controlled HTML files, and no ongoing infrastructure costs. As an additional finding, the artifact was verified to run without modification on a smartphone (Samsung Galaxy S24, mobile Chrome), a direct consequence of the zero-dependency model and native WebAssembly support in mobile browsers, confirming that the edge execution model extends naturally to any standards-compliant device.

References

Chen, X., Lin, M., Schärli, N., & Zhou, D. (2024). Teaching large language models to self-debug. In Proceedings of the 12th International Conference on Learning Representations (ICLR 2024). https://openreview.net/forum?id=KuPixIqPiq

Corlay, S. (2025, March 12). Teaching a billion people to code: How JupyterLite is scaling the impossible. The New Stack. https://thenewstack.io/teaching-a-billion-people-to-code-how-jupyterlite-is-scaling-the-impossible/

Droettboom, M. (2018). Scientific Python in the browser [Blog post]. Droettboom. https://droettboom.com/blog/2018/04/04/python-in-the-browser/

Drobnjak, A., & Boticki, I. (2025). Modelling virtual educational agents: Challenges of realism, performance and quality of audiovisual content. In R. A. Sottilare & J. Schwarz (Eds.), Adaptive instructional systems. HCII 2025. Lecture Notes in Computer Science, vol. 15813 (pp. 180–189). Springer. https://doi.org/10.1007/978-3-031-92970-0_13

Fink, M. C., Robinson, S. A., & Ertl, B. (2024). AI-based avatars are changing the way we learn and teach: Benefits and challenges. Frontiers in Education, 9, 1416307. https://doi.org/10.3389/feduc.2024.1416307

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., & Bastien, J. F. (2017). Bringing the web up to speed with WebAssembly. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017) (pp. 185–200). ACM. https://doi.org/10.1145/3062341.3062363

Jangda, A., Powers, B., Berger, E. D., & Guha, A. (2019). Not so fast: Analyzing the performance of WebAssembly vs. native code. In Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19) (pp. 107–120). USENIX Association. https://www.usenix.org/conference/atc19/presentation/jangda

Keuning, H., Jeuring, J., & Heeren, B. (2018). A systematic literature review of automated feedback generation for programming exercises. ACM Transactions on Computing Education, 19(1), 1–43. https://doi.org/10.1145/3231711

Langa, Ł. (2025, February). A peek into a possible future of Python in the browser [Blog post]. https://lukasz.langa.pl/f37aa97a-9ea3-4aeb-b6a0-9daeea5a7505/

Letteri, I., & Vittorini, P. (2025). Enhancing student feedback in data science education: Harnessing the power of AI-generated approaches. International Journal of Artificial Intelligence in Education, 35, 3071–3094. https://doi.org/10.1007/s40593-025-00492-8

Lee, S. S., & Moore, R. L. (2024). Harnessing generative AI for automated feedback in

higher education: A systematic review. Online Learning, 28(3), 82–104. https://doi.org/10.24059/olj.v28i3.4593

Liu, Q., Zuo, Y., Ye, R., Chang, Y., Zhang, X., & Wu, L. (2026). Design and development of an LLM-empowered arguing pedagogical agent with a personalized avatar, movements, and voice. In Q. Liu et al. (Eds.), Educational Innovation Through Technology. EITT 2024. Communications in Computer and Information Science, vol 2600. Springer. https://doi.org/10.1007/978-981-95-1525-7_2

Ma, Q., Shen, H., Koedinger, K., & Wu, S. T. (2024). How to teach programming in the AI era? Using LLMs as a teachable agent for debugging. In Proceedings of the 25th International Conference on Artificial Intelligence in Education (AIED 2024). Lecture Notes in Computer Science, vol 14829. Springer. https://doi.org/10.1007/978-3-031-64302-6_19

Pimentel, J. F., Murta, L., Braganholo, V., & Freire, J. (2019). A large-scale study about quality and reproducibility of Jupyter notebooks. In Proceedings of the 16th IEEE/ACM International Conference on Mining Software Repositories (MSR 2019) (pp. 507–517). IEEE. https://doi.org/10.1109/MSR.2019.00077

Satyanarayanan, M., Simoens, P., Xiao, Y., Pillai, P., Chen, Z., Ha, K., Hu, W., & Amos, B. (2015). Edge analytics in the Internet of Things. IEEE Pervasive Computing, 14(2), 24–31. https://doi.org/10.1109/MPRV.2015.32

Scholz, N., Nguyen, M. H., Singla, A., & Nagashima, T. (2025). Partnering with AI: A pedagogical feedback system for LLM integration into programming education. arXiv:2507.00406. https://arxiv.org/abs/2507.00406

Shi, W., Cao, J., Zhang, Q., Li, Y., & Xu, L. (2016). Edge computing: Vision and challenges. IEEE Internet of Things Journal, 3(5), 637–646. https://doi.org/10.1109/JIOT.2016.2579198

Taylor, A., Vassar, A., Renzella, J., & Pearce, H. (2024). dcc --help: Transforming the role of the compiler by generating context-aware error explanations with large language models. In Proceedings of the 55th ACM Technical Symposium on Computer Science Education (SIGCSE 2024) (Vol. 1, pp. 1–7). ACM. https://doi.org/10.1145/3626252.3630822

Yang, B., Tian, H., Pian, W., Yu, H., Wang, H., Klein, J., Bissyandé, T. F., & Jin, S. (2024). CREF: An LLM-based conversational software repair framework for programming tutors. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024) (pp. 882–894). ACM. https://doi.org/10.1145/3650212.3680328