In-Browser C++ Interpreter for Lightweight Intelligent Programming Learning Environments

Tomas Blažauskas
Department of Software Engineering, Kaunas University of Technology, Lithuania

Arnoldas Rauba
Department of Software Engineering, Kaunas University of Technology, Lithuania

Jakub Swacha
Department of Information Technology in Management, University of Szczecin, Poland

Raffaele Montella
Department of Science and Technology, University of Naples "Parthenope", Italy

Rytis Maskeliunas
Department of Software Engineering, Kaunas University of Technology, Lithuania

Abstract

The paper presents a browser native C++ interpreter integrated into an AI-assisted educational platform designed to enhance programming learning in formal education. The interpreter leverages Parsing Expression Grammars (PEG) to generate Abstract Syntax Trees (AST) and executes C++ code using a TypeScript-based runtime. The system supports key C++ features, including pointer arithmetic, function overloading, and namespace resolution, and emulates memory management via reference-counted JavaScript objects. Integrated within a web-based learning environment, it provides automated feedback, error explanations, and code quality evaluations. The evaluation involved 4582 students in three difficulty levels and feedback from 14 teachers. The results include high system usability scale (SUS) scores (avg. 83.5) and WBLT learning effectiveness scores (avg. 4.58/5). Interpreter performance testing in 65 cases averaged under 10 ms per task, confirming its practical applicability to school curricula. The system supports SCORM and PWA deployment, enabling LMS-independent usage. The work introduces a technical innovation in browser-based C++ execution and a scalable framework for LLM-enhanced programming pedagogy.

2012 ACM Subject Classification Software and its engineering → General programming languages

Keywords and phrases C++ interpreter, browser-based execution, programming education, LLM-assisted learning, PEG, AST, TypeScript runtime

Digital Object Identifier 10.4230/OASIcs.ICPEC.2025.14

Funding This research was funded by the Erasmus programme under the grant number 2023-1-PL01-KA220-HED-000164696.

Acknowledgements We want to thank Renata Burbaitė and KTU students for the creation of educational resources at open.ktu.edu that were used in this research.

1 Introduction

The integration of artificial intelligence (AI) and especially the rapid progress in the capabilities of large language models (LLMs) in educational technologies has revolutionized programming pedagogy, allowing automated error detection, code correction, and real-time feedback mechanisms [26]. Recent advancements in browser-based interpreters, such as Brython for Python and JS-Interpreter for JavaScript, have demonstrated the feasibility of executing lightweight, interpreted languages in web environments, fostering interactive coding platforms. However, extending such capabilities to compiled languages like C++ remains a significant challenge. Our work addresses the critical gap in existing frameworks

by proposing a novel execution and evaluation environment featuring a C++ interpreter engine designed for serverless execution in browsers, in particular, to be coupled with the FGPE platform [25].

Current approaches to web-based C++ execution, such as WebAssembly-based compilers (e.g., Emscripten) or transpilers like Cheerp, prioritize complete compilation over interactive interpretation, limiting their utility for real-time feedback in educational settings. Furthermore, AI integration in programming education has focused predominantly on high-level languages, with fewer efforts addressing the idiosyncrasies of systems programming paradigms. Our research uses a hybrid approach: we use Parsing Expression Grammars (PEG) via peg.js to parse C++ source code into an Abstract Syntax Tree (AST), which is evaluated in a TypeScript runtime. By implementing core C++ features, including pointer arithmetic, function overloading, and namespace resolution, within a browser environment, the proposed engine bridges the divide between traditional desktop development and modern web-native tooling.

Integration into the FGPE framework [25] further extends this infrastructure by providing access to gamified programming exercises and their automatic evaluation with meaningful feedback to students. We believe this well aligns with trends in intelligent tutoring systems (ITS) that use static and dynamic analysis to personalize learning [19]. However, unlike ITS platforms targeting Python or Java, our solution uniquely accommodates the dual role of C++ as both a teaching language and a systems programming tool. By decoupling exercise generation from Learning Management Systems (LMS) and embracing serverless execution, the framework ensures scalability and accessibility, critical for widespread adoption in educational ecosystems.

2 State of the art review

The execution of compiled languages like C++ in browser environments remains a niche yet critical domain, contrasting sharply with the maturity of interpreted language implementations (e.g., Python via Pyodide or Brython). Contemporary research focuses primarily on two paradigms: WebAssembly (Wasm) based compilation [36] and AST-driven interpreters [15]. Wasm frameworks such as Emscripten [8] excel in performance, allowing for near-native execution by translating C++ binaries into Wasm modules. However, their sandboxed runtime obscures runtime introspection (ability of a program to examine its own state, structure, and execution flow while it is running), limiting its utility in pedagogical contexts that require granular error tracking or step-by-step debugging [27]. In contrast, AST-based interpreters, such as those that underpin JavaScript-in-JavaScript systems (e.g., JS-Interpreter), prioritize transparency over speed, allowing real-time code behavior analysis [1].

A second frontier lies in runtime variable management and type simulation [31], particularly for low-level languages like C++. Systems such as Pyodide [24] use Python's dynamic typing to abstract memory management, bypassing the complexities inherent in C++. In contrast, C++ interpreters must deal with manual memory allocation, pointer arithmetic, and implicit type conversions [32]. Other approaches, including Grasp.js [35] AST analyzer and Microsoft SAL annotations, employ type-checking frameworks to enforce safety guarantees in JavaScript-hosted runtimes.

Finally, integrating AI-driven code evaluation and exercise generation represents a nascent but transformative direction [7, 6]. While tools like GitHub Copilot [10] and DeepCode [29] focus on code suggestion in IDEs, browser-hosted systems face stricter latency and sandboxing constraints. Intelligent tutoring systems (ITS) use static analysis for feedback, but lack a

dynamic context-aware error explanation [20]. Transformer-based models (e.g., Codex [16] or AlphaCode [18]) hint at possibilities for autonomous exercise generation, yet their deployment in browser environments remains unexplored.

Beyond solving exercises, LLMs have been applied to the automatic creation of programming exercises. Approaches involving autonomous agents [13, 21] show that these models can generate high-fidelity content by capturing probabilistic links between programming constructs and their associated natural language instructions. The underlying transformer-based architectures [12] excel at modeling both contextual depth and long-term dependencies, allowing the generation of exercises that are both technically precise and pedagogically targeted [17]. Modern implementations often involve model fine-tuning with narrowly focused datasets, such as annotated educational repositories or competition-grade coding challenges. This process improves both content relevance and difficulty regulation. Advances in quick-fire learning techniques and prompt optimization further enable these systems to accommodate various instructional scenarios [5]. Their applicability spans from beginner programming education to niche areas like machine learning algorithms and distributed computing. Some architectures have also been customized to support gamified learning platforms, such as FGPE [22].

However, despite the promising progress, several unresolved issues remain. One major challenge is the presence of inherent bias in LLM training data [33], which can lead to unequal representation between topics, fluctuating difficulty levels, and a lack of cultural inclusion. Mitigating these biases will require the development of algorithmic correction techniques [34] and the creation of diversified training corpora through collaborative and interdisciplinary input. Another concern is randomness in LLM outputs [23], which can cause variability in exercise quality and assessment accuracy, thus affecting consistency and repeatability in educational applications.

3 Materials and methods

3.1 Dataset

To validate the proposed solution, we use open-access courses created by our team and provided at open.ktu.edu. The courses contain materials for three different difficulty levels: Programming Lessons, level 1 (C++), Programming Lessons, level 2 (C++), and Programming Lessons, level 3 (C++). During the last school year (2024-2025), the courses were used by 4582 learners (3443 in course 1, 731 in course 2, and 408 in course 3).

The first course contains 67 programming exercises grouped into the following topics: linear algorithms, branched algorithms, loops with known iteration number, loops with unknown iteration number, calculation of element quantity, sum, and average, nested loops, and frequently encountered programming exercises. In addition to programming exercises, the course contains 90 executable code snippets to support the theory material. The executable snippets are created with the tool used to develop programming exercises. The example of executable snippets is shown in Fig. 1.

The second course contains 44 exercises grouped into the following topics: reading from and writing to files; a function that returns a computed value via the function name; a function with reference parameters; static arrays (calculation of quantity, sum, and average); static arrays (minimal and maximal elements); static arrays (search, insertion, and removal); sorting algorithms; and dynamic arrays using the vector library. This course contains 74 executable code snippets to support the theory material.

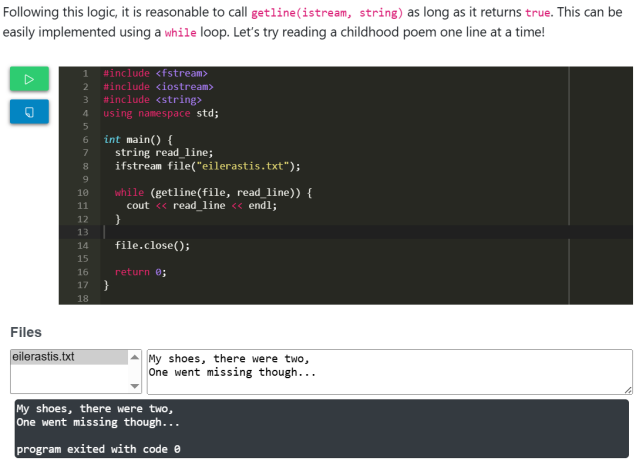

Figure 1 The example of the executable snippet to support theory.

The third course contains 20 exercises. The third course contains considerably fewer programming exercises because only a few new programming concepts were introduced to learners in the last school year. The course includes the following topics: advanced string usage and char arrays; structures. There is also testing material, but it does not contain programming exercises (just executable snippets to support theory). In total, there are 23 executable snippets.

3.2 Implementation

The FGPE Lite environment is sketched in Fig. 2. Compared to the original FGPE environment [25], it uses a lightweight server only to store gamified programming exercises and student progress. On the client side, the FGPE web app downloads gamified exercises and student progress from a web server. It also communicates with a web server to store student progress. The minimal communication can be postponed until the internet connection is available. Besides UI, which implements a code editor, result and achievement visualization, the client side also includes a gamification engine to monitor the students' actions and generate instant feedback without a need to consult the server [30]. The execution environment runs student code in a browser environment. The web server also communicates with the learning management system to store student progress in a course. The LTI (Learning Tool Interoperability) standard is used for that purpose.

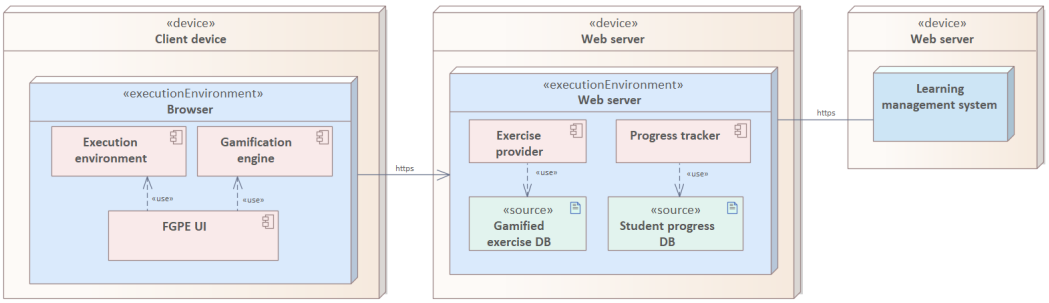

Figure 2 FGPE Lite environment.

Although the FGPE framework features the AuthorKit exercise editor, deeply embedded in the original FGPE software stack [25], with a recently added layer of automatic gamified exercise generation based on LLM [22], for the sake of this research, we decided to develop and use an autonomous light-weight programming exercise creation, execution and testing tool. In later development stages, we consider fully integrating it with the FGPE ecosystem as a lighter alternative to AuthorKit. The composition of the programming exercise creation tool is shown in Fig. 3. The main component is a *Programming task builder*. It uses *Task creation form* to collect information about the programming task. The creation tool can export SCORM (Sharable Content Object Reference Model) and PWA (Progressive Web App) packages. The SCORM package is a `.zip` archive designed for use with leaA printable version of an exercise can also SCORM template directory is updated using information from the *Task creation form* to produce the final *programming_task.zip*. The *PWA template* includes additional files to facilitate the creation of an installable web application, but the resulting archive remains the same *programming_task.zip*. Both templates use the *cpp_interpreter.js* file, which implements C++ code interpretation. Additionally, a printable version of an exercise can be generated as a PDF document (*programming_task.pdf*). It is also possible to publish the programming task as a link; for that purpose, the tool creates a *programming_task_dir* directory, uniquely identified by the task ID.

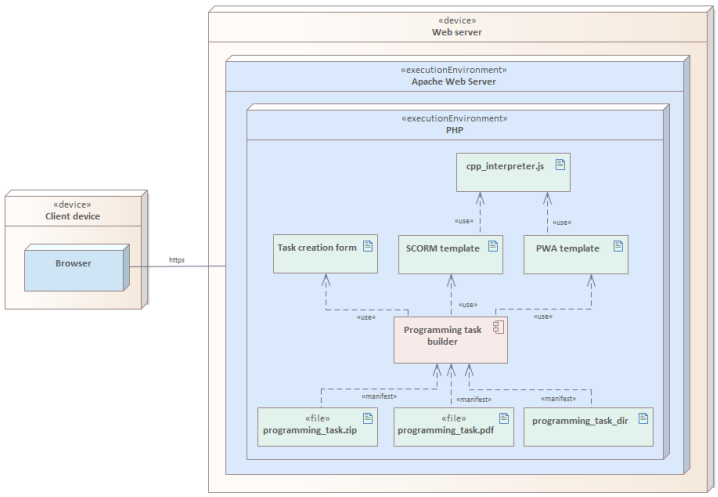

Figure 3 Programming task creation tool composition.

The execution and evaluation environment is produced by the programming task creation tool. It is downloaded as a *programming_task.zip*. It can be extracted and executed on a local machine or uploaded to a SCORM standard-supported learning management system.

The *Hello, world!* programming task is shown in Fig. 4.

The composition of the execution and evaluation environment is shown in Fig. 5. This scenario is as if the generated solution is used as a PWA (Progressive Web App). The *index.html* file contains the structure of the execution and evaluation environment. It uses *style.css* for element design and *utilities* for various supporting functionality. For example, it includes the ACE code editor for editing program code. The *cpp_ide.js* is a wrapper for *cpp_interpreter.js*. It handles the input from and output to the console. It handles writing to and reading from files. We use the virtual (in-memory) file system for that purpose. Each file is created as a separate node in the DOM (Document Object Model). Also, it handles

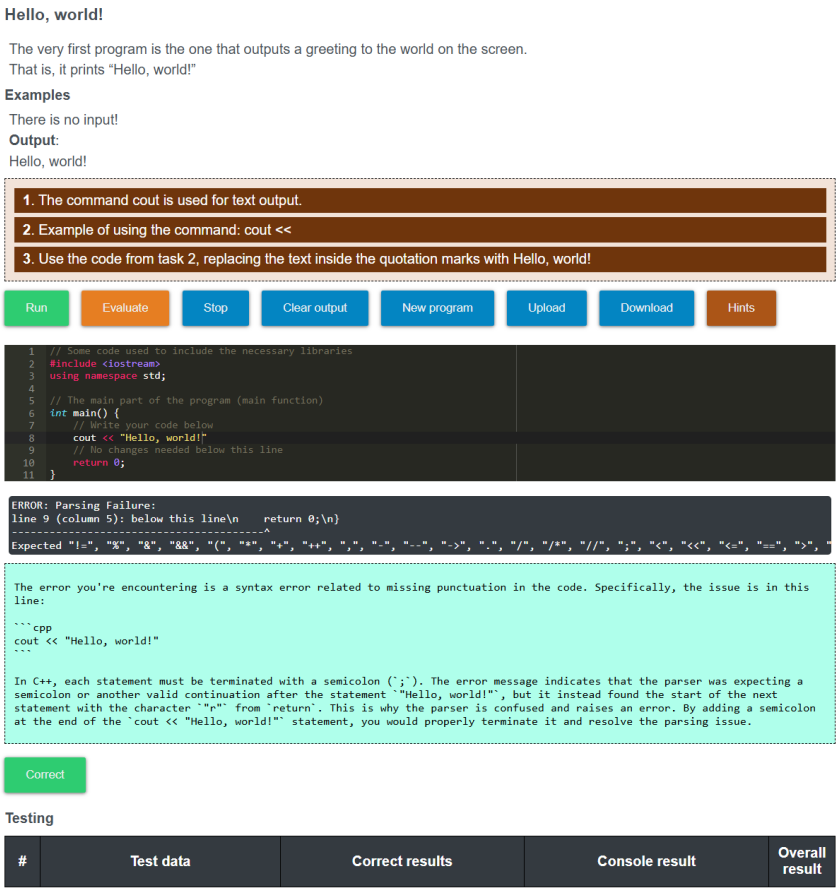

Figure 4 Execution and evaluation environment.

exceptions thrown by the interpreter. The wrapper also prepares test cases (from the file *test.xml* for evaluation. Finally, *cpp$_i$nterpreter.js* is the C++ interpreter that interprets the code provided by *cpp$_i$de.js*, taken from the *index.html* code editor.

3.2.1 C++ interpreter implementation

Initially, we started the implementation of the interpreter from the Felix Hao open source project [11]. This project has a very good foundation, although it lacks the functionality required for school curricula. We extended this engine to cover most language features required for school curricula. The forked and extended engine is located here [9]. We added support for structures and file operations. Also, we implemented vector, algorithm, string, sstream libraries. Finally, we rewrote the interpreter to enable advanced C++ features such as function overloading and dynamic memory management.

The CPP interpreter engine implements the runtime to execute the C++ source code in a web browser environment. The given C++ source code is parsed and then passed to the Abstract Syntax Tree (AST). The parser and pre-processor are created using the peg.js Parsing Expression Grammars (PEG) parser generator. The AST walker exports the function definitions, including the entry point, main(), to the runtime, and then evaluates the main function. The execution of the code ends after the AST walker approaches the end of the main function or a run-time error.

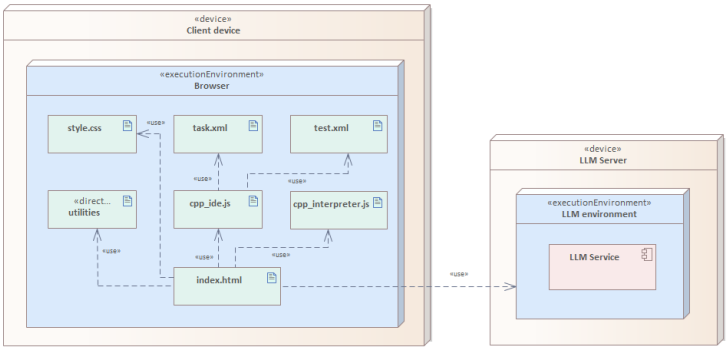

Figure 5 Execution and evaluation environment components.

The interpretation workflow is presented in Fig. 6. The C++ interpreter traverses and evaluates the Abstract Syntax Tree (AST) during the program's execution. The evaluation begins with the initialization of an AST walker, starting from the root node of the tree, which is typically the compound statement node that contains all the instructions of the function. The interpreter enters a recursive depth-first-search traversal of the AST, implemented as a JavaScript generator function.

Each node encountered's type is identified and handled accordingly through a type-specific dispatch mechanism. For instance, when encountering a function or operator call, the interpreter attempts to match it with a defined function signature, pushes a new value scope onto the stack to isolate local variables, and invokes the generator function associated with the defined function. Literal values create non-lvalue (temporary) variables, while variable declarations lead to the allocation of lvalue variables within the current scope. Return statements are handled by bringing the returned value to the top of the evaluation context and ignoring the evaluation of further statements.

The given list of node types is not exhaustive and consists of other node types that correspond to certain expressions of the C++ programming language. The execution of the global event loop can be paused when user input is required (e.g., reading a line from the standard input), terminated when approaching a runtime error, or triggered by the user to stop the execution. This traversal continues iteratively until the stack of nodes becomes empty, at which point the AST evaluation concludes and the interpreter exits.

The C++ standard library functions are implemented internally in TypeScript code. However, sufficient progress has been made to limit the number of functions that can only be implemented internally to a minimum. Every include library contains the loading function, which registers the data structures and functions. Partial support for the C++ namespaces is implemented.

The complex nature of C++ introduces several technical challenges when writing a compiler or an interpreter. These include implicit cast of a structure type to a boolean, function parameter overloads, and multiple variable value types. They also include the heritage from the C programming language, such as manual memory management, unsafe pointer casts, and implicit conversions between arithmetic types. Several of these issues are addressed in the runtime to ensure the correctness of the input C++ code and detect commonly made mistakes. Runtime variables are implemented as JavaScript objects holding a value object and a type object, allowing multiple variables to borrow and modify the same l-value, as in standard C++. Of all possible C++ types, the engine supports arithmetic

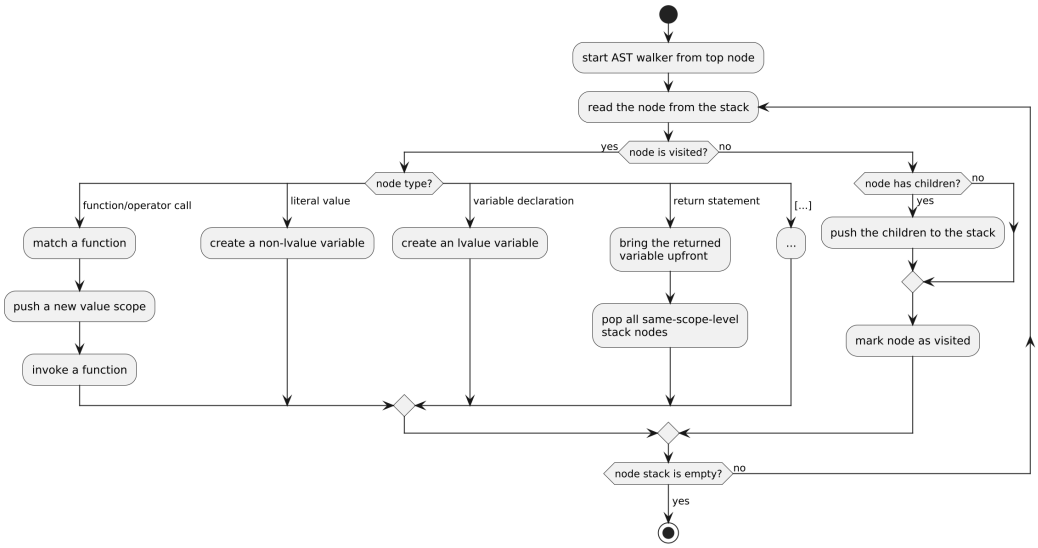

Figure 6 Code interpretation loop.

types, pointers, compound struct/class types, and function types. Functions are not variables but can be pointed to by a pointer variable, as in standard C++. A type object may contain a member of another type, as with pointer types.

4 Experimental evaluation

To evaluate the tool as a learning object, a WBLT scale (Web-Based Learning Tools) scale [14] was used to assess the constructs of learning, design, and participation. The SUS (System Usability Scales) scales [4] were used to evaluate the tool's usability. In addition to these standard metrics, specific questions related to the tool's functionality, AI-assisted support, and the programming task-solving process were included. In addition, interpreter performance was evaluated by testing its execution speed with real C++ programming exercises. Fourteen school teachers participated in a survey using open-access courses at open.ktu.edu (see section 3.1 for detailed description).

4.1 Evaluation of a solution as a learning tool

The evaluation scores for the WBLT components were obtained from survey data by calculating the average of the ratings for the corresponding items. As shown in Fig. 7, the average scores for the WBLT subscales were high. It was expected, as the tool was developed intensively with the close collaboration of Lithuanian teachers. Internal consistency was evaluated using Cronbach's Alpha. The Learning ($\alpha = 0.873$) and Design ($\alpha = 0.883$) subscales demonstrated good reliability, while the Engagement subscale showed acceptable reliability ($\alpha = 0.714$).

The Learning subscale received a high average score of 4.58 (standard deviation: 0.62), indicating that most teachers positively evaluated the tool's contribution to the learning process. The scores ranged from 3.2 to 5, while more than half of the respondents gave the highest possible rating in this category. The Design subscale had a slightly lower average of 4.25, with a standard deviation of 0.88 - higher than for the other subscales. This suggests a greater variation in user opinions about the design. The scores ranged from 2 to 5, but the

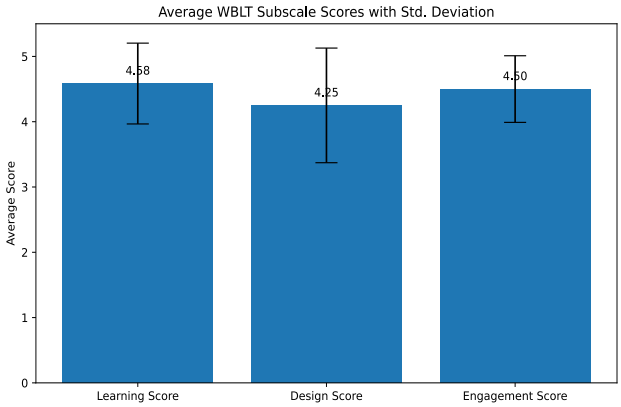

Figure 7 WBLT evaluation results.

median was still high at 4.5, and 75% of the respondents gave a score of 5. The Engagement subscale was highly positively rated, with an average score of 4.5 (standard deviation: 0.51), and all scores fell within a relatively narrow range of 3.5 to 5. This indicates that users felt engaged while using the tool and had a positive learning experience.

Consistently high scores across all WBLT subscales have shown the pedagogical robustness of the tool and its alignment with evidence-based instructional design principles. In particular, the strong performance in the learning dimension (M = 4.58, SD = 0.62) indicates that the tool effectively supports cognitive engagement and knowledge construction, fulfilling its core educational function. The narrow range and high-rated concentration further reflect a shared perception among educators that the tool contributes meaningfully to learning outcomes. Its co-development with Lithuanian teachers ensured curricular alignment, contextual relevance, and the integration of pedagogical advantages tailored to local instructional needs. Such collaborative development practices are aligned with the participatory design framework used in programming engineering courses and have been shown to enhance the adoption and sustainability of digital learning tools in formal education settings.

4.2 Tool usability evaluation

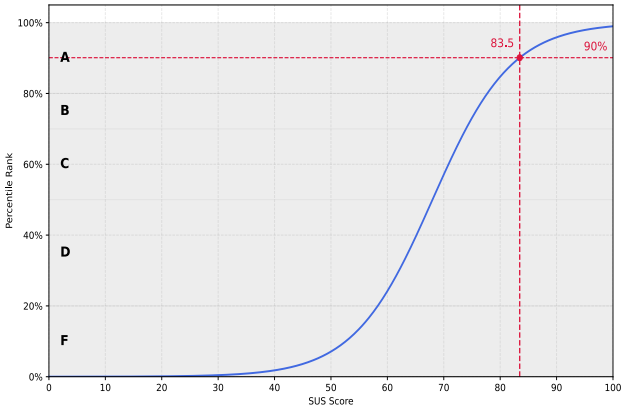

Figure 8 SUS evaluation results.

The System Usability Scale (SUS) score of 83.5 places the tool within the upper echelon of user-centered design performance, reflecting a strong consensus among users on its usability and intuitiveness. When normalized against benchmark SUS datasets, this score corresponds to approximately the 90th percentile, indicating that the system outperforms comparable tools 90% in perceived usability. This percentile rank not only confirms the technical adequacy of the system but also suggests that its interface, workflow logic, and interaction design are closely aligned with user expectations and established human-computer interaction (HCI) heuristics. This level of usability is particularly noteworthy given the diversity of users typically involved in educational settings, where tools must accommodate a wide range of digital competencies. The internal consistency of the SUS scale, measured using Cronbach's alpha, was 0.745, indicating an acceptable level of reliability. This shows that the 10 SUS elements coherently reflect a unidimensional construct of perceived usability and that the resulting composite scores can be interpreted as valid usability indicators.

From a psychometric and interpretative point of view, the SUS score of 83.5 corresponds to an "A" grade within the curved grading framework proposed by Sauro and Lewis [28], which indicates good usability relative to industry norms. The grading system accounts for the nonlinear nature of user satisfaction and offers a nuanced classification of usability levels based on normalized score distributions. Furthermore, the tool's SUS score is located in the upper bound of the "Good" adjective rating and approaches the "Excellent" category established by Bangor et al. [2], reinforcing the validity of its favorable reception. The acceptability framework, which designates scores above 70 as unequivocally acceptable, further underscores the system's readiness for wide-scale adoption in formal educational contexts.

4.3 Evaluation of interpreter and AI features

To evaluate the C++ interpreter and AI features used in our tool, specific questions were constructed aimed at the coverage and performance of the C++ language (questions Q1-Q4) and the three AI functions we use within our tool: error explanation, code correction, and code quality evaluation (questions Q5-Q9).

Table 1 summarizes the teacher responses to nine survey items on the C++ interpreter and AI-assisted features in the programming environment. The first four questions focus on interpreter capabilities. The results indicate that the programming tasks align well with the national school curriculum (Q1), receiving a strong normalized average of 0.86. In particular, all teachers positively rated the execution speed (Q2, normalized to 1.00), confirming that performance is sufficient for educational purposes. Although Q3 results suggest minimal perceived discrepancies with full C++ implementations (average: 0.92), it is crucial to note that this perception may stem from the limited subset of language features used in schools: Our interpreter remains substantially less compliant with the complete standard. The relatively lower score for Q4 (0.78) reflects concerns about the informativeness of native error messages, probably motivated by the integration of AI for improved feedback clarity.

The remaining five questions relate to the AI assistant's error support and code evaluation role. Teachers strongly believed in the necessity of AI-assisted error explanations (Q5, score: 0.96), and they generally agreed on their usefulness (Q6: 0.89), although some variation in perceived quality was noted. Opinions diverged more significantly regarding the AI's role in automated code correction (Q7: 0.76), with some educators cautioning that excessive automation may hinder student learning. Based on this feedback, the feature was made optional and is currently disabled by default for learners. Finally, most respondents endorsed

Table 1 Survey results for C++ interpreter and AI assistant features.

ID	Survey Question	Normalized Avg. Score
Q1	The programming tasks cover the school curriculum.	0.86
Q2	The program code for the given tasks runs fast enough.	1.00
Q3	There are discrepancies between the "real" C++ language and this implementation.	0.92
Q4	The error messages (without AI assistance) are sufficiently informative.	0.78
Q5	I believe that AI assistance explaining the error messages is necessary.	0.96
Q6	The AI-generated explanations of error messages are appropriate.	0.89
Q7	I believe that the AI-based code correction feature is necessary.	0.76
Q8	I believe that the AI-based code quality assessment feature is necessary.	0.89
Q9	I believe that the code quality assessment is informative and appropriate.	0.82

AI-driven code quality evaluation in terms of necessity (Q8: 0.89) and informativeness (Q9: 0.82), although some variance suggests a need for refinement in feedback granularity and pedagogical tone.

4.4 Evaluation of C++ interpreter performance

One of the most remarkable properties of interpreters is their speed or performance. In an educational context, performance is not as important; therefore, our idea was to evaluate how this interpreter performs with the code required to solve various programming tasks. We decided to use our automated integration tests because these tests are based on programming tasks designed to cover most of the interpreter functionality. The integration tests are available in this repository [3].

There are 17 integration tests to test 17 programming tasks. Each programming task has up to six testing cases (inputs and appropriate outputs to check on the same problem). In total, 65 test cases were used for the evaluation. Each test case was run 50 times to avoid possible result dissipation due to side processes. The performance of the given test cases was measured on an Intel i7-1260P CPU, 16 GB of RAM, on a 64-bit Linux 6.11.0-114024-tuxedo kernel, using Node.js version v22.14.0. The results are summarized in Table 2. Some names of the problems in Lithuania were left to match those in the provided repository. The average duration for all problems is less than 10 milliseconds. The maximum execution time was 48 milliseconds. This test showed that the interpreter's performance is good enough for many school-level programming tasks. It supports the subjective evaluation of teachers' perceived performance. Still, our future work includes optimizing our interpreter to solve more complex problems that can be encountered in various programming competitions.

5 Discussion and Conclusions

The intersection of AI-driven code interpretation and browser-based execution environments has emerged as a critical domain in computational education, particularly for languages such as C++ that pose unique implementation challenges in web contexts. Traditional approaches for interpreted languages (e.g., Python via Brython, Skulpt, or Pyodide) leverage existing JavaScript engines and transpilation techniques to achieve browser compatibility. However, compiled languages such as C++ require fundamentally different strategies due to their reliance on static typing, manual memory management, and low-level constructs. Recent advancements in WebAssembly (Wasm) have enabled near-native execution of compiled

Table 2 Execution times of C++ integration tests.

Nr.	Problem name	Average duration (ms)	max Duration (ms)
1	atom	17.10	48
2	dalelė	6.85	15
3	detalės	10.08	17
4	dėžės	4.09	8
5	gyventojai	6.98	13
6	kopija	1.59	3
7	melaginga-zinia	7.90	45
8	muziejus	30.35	43
9	rudeninė_mugė	4.18	6
10	sodinukai	6.34	9
11	sort_by_comparator	9.04	13
12	sort_static_array	1.74	3
13	stod_stof_stol_stoi	2.55	4
14	string_preprocessing	1.44	2
15	subjects	7.00	10
16	vector_subjects	7.62	12
17	vėlavimas	10.35	19

code via tools like Emscripten, which compiles C++ to Wasm for sandboxed execution in browsers. Although effective, these methods often sacrifice run-time introspection and dynamic error analysis, key requirements for educational frameworks. The proposed FGPE C++ interpreter engine addresses this gap by combining Parsing Expression Grammars (PEG) with Abstract Syntax Tree (AST) traversal, enabling real-time code evaluation and error diagnosis without relying on external compilation pipelines and whole FGPE approach aligns with emerging trends in pedagogical tools that prioritize interactive feedback over raw performance. However, it introduces trade-offs in handling complex C++ features like pointer arithmetic and template meta-programming, which remain poorly supported in existing browser-based interpreters.

Integrating explainable AI in the feedback loop – through symbolic execution, AST diffing, and constrained natural language explanations – shifts the paradigm from correction to conceptual clarity. Unlike IDE-based LLM integrations prioritizing productivity, this system foregrounds learning outcomes by deliberately limiting automation (for example, code correction is optional) in favor of scaffolded reasoning. The strong correlation between high engagement scores and the perceived necessity of error explanation (Q5: 0.96) underscores that learners value interpretability over mere correctness.

Perhaps most importantly, the findings challenge a common assumption in programming education: only interpreted or high-level languages are suitable for LLM-supported interactive environments. By demonstrating sub-10ms execution latency for real C++ tasks and delivering a SUS score in the 90th percentile, this work reveals that performance–observability trade-offs are surmountable for systems languages when architectural decisions privilege pedagogical alignment over raw throughput.

The results position the interpreter not only as a technical artifact but as a bridge between formal semantics, instructional design, and AI-enhanced learning ecosystems. Future extensions, such as deeper template support, richer namespace modeling, and formal verification backends, could refine this framework to scale from high-school instruction to undergraduate systems programming and competitive coding education. Its SCORM/PWA-compatible

deployment also opens avenues for integration into decentralized, learner-owned educational infrastructures, a prospect aligned with the growing push for open, explainable, and portable EdTech tools.

However, while the current implementation demonstrates feasibility, future work will further address interoperability with established LMS ecosystems (e.g., Moodle, Canvas) and validate pedagogical efficacy through longitudinal studies. In these areas, Python-centric platforms like JupyterHub have set precedents.

References

1 Ch Aneesh, Gembali Saumik, Kvv Varun, and Meena Belwal. Smart compiler assistant: An ast based python code analysis. In *2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–6. IEEE, 2024.

2 Aaron Bangor, Philip T. Kortum, and James T. Miller. Determining what individual sus scores mean: Adding an adjective rating scale. *Journal of Usability Studies*, 4(3):114–123, 2009.

3 Tomas Blažauskas. Jscpp integration tests. `https://github.com/blaztoma/JSCPP/tree/master/integration-tests`, 2025. Accessed: 2025-05-17.

4 John Brooke. *SUS – a quick and dirty usability scale*, pages 189–194. Taylor and Francis, January 1996.

5 William Cain. Prompting change: exploring prompt engineering in large language model ai and its potential to transform education. *TechTrends*, 68(1):47–57, 2024.

6 Doga Cambaz and Xiaoling Zhang. Use of ai-driven code generation models in teaching and learning programming: a systematic literature review. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, pages 172–178, 2024.

7 Liguo Chen, Qi Guo, Hongrui Jia, Zhengran Zeng, Xin Wang, Yijiang Xu, Jian Wu, Yidong Wang, Qing Gao, Jindong Wang, et al. A survey on evaluating large language models in code generation tasks. *arXiv preprint arXiv:2408.16498*, 2024.

8 Emscripten Contributors. Main — Emscripten 4.0.9-git (dev) documentation. `https://emscripten.org/`. [Accessed 15-05-2025].

9 Tomas Blažauskas Felix Hao. Jscpp: Enhanced c++ interpreter for educational use. `https://github.com/blaztoma/JSCPP`, 2025. Accessed: 2025-05-17.

10 GitHub. Github copilot - your ai pair programmer. `https://github.com/features/copilot`. [Accessed 15-05-2025].

11 Felix Hao. JSCPP: A c++ interpreter in javascript. `https://github.com/felixhao28/JSCPP`, 2015. Accessed: 2025-05-17.

12 Dawei Huang, Chuan Yan, Qing Li, and Xiaojiang Peng. From large language models to large multimodal models: A literature review. *Applied Sciences*, 14(12):5068, 2024.

13 Hyoungwook Jin, Seonghee Lee, Hyungyu Shin, and Juho Kim. Teach ai how to code: Using large language models as teachable agents for programming education. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–28, 2024.

14 Robin Kay. Evaluating learning, design, and engagement in web-based learning tools (wblts): The wblt evaluation scale. *COMPUTERS IN HUMAN BEHAVIOR*, 27(5):1849–1856, September 2011. `doi:10.1016/j.chb.2011.04.007`.

15 Octave Larose, Sophie Kaleba, Humphrey Burchell, and Stefan Marr. Ast vs. bytecode: interpreters in the age of meta-compilation. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):318–346, 2023.

16 Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning, 2022. `doi:10.48550/arXiv.2207.01780`.

17 Qingyao Li, Lingyue Fu, Weiming Zhang, Xianyu Chen, Jingwei Yu, Wei Xia, Weinan Zhang, Ruiming Tang, and Yong Yu. Adapting large language models for education: Foundational capabilities, potentials, and challenges. *arXiv preprint arXiv:2401.08664*, 2023.

18 Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. doi:10.1126/science.abq1158.

19 Chien-Chang Lin, Anna YQ Huang, and Owen HT Lu. Artificial intelligence in intelligent tutoring systems toward sustainable education: a systematic review. *Smart Learning Environments*, 10(1):41, 2023.

20 Vincent Liu, Ehsan Latif, and Xiaoming Zhai. Advancing education through tutoring systems: A systematic literature review. *arXiv preprint arXiv:2503.09748*, 2025.

21 Qianou Ma, Hua Shen, Kenneth Koedinger, and Sherry Tongshuang Wu. How to teach programming in the ai era? using llms as a teachable agent for debugging. In *International Conference on Artificial Intelligence in Education*, pages 265–279. Springer, 2024.

22 Raffaele Montella, Ciro Giuseppe De Vita, Gennaro Mellone, Tullio Ciricillo, Dario Caramiello, Diana Di Luccio, Sokol Kosta, Robertas Damaševičius, Rytis Maskeliūnas, Ricardo Queiros, and Jakub Swacha. Leveraging Large Language Models to Support Authoring Gamified Programming Exercises. *Applied Sciences*, 14(18):1–15, 2024. doi:10.3390/app14188344.

23 Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. Llm is like a box of chocolates: the non-determinism of chatgpt in code generation. *arXiv preprint arXiv:2308.02828*, 2023.

24 Pyodide Contributors. Pyodide 2014; version 0.27.5 - pyodide.org. https://pyodide.org/en/stable/. [Accessed 15-05-2025].

25 Ricardo Queirós, Jakub Swacha, Robertas Damasevicius, and Rytis Maskeliunas. Fgpe-an evolving framework for gamified programming learning. In *International Conference on Advanced Research in Technologies, Information, Innovation and Sustainability*, pages 313–323. Springer, 2024.

26 Venushini Rajendran and R Kanesaraj Ramasamy. Artificial intelligence integration in programming education: Implications for pedagogy and practice. In *International Conference on Computer Vision, High-Performance Computing, Smart Devices, and Networks*, pages 197–206. Springer, 2023.

27 Alan Romano, Xinyue Liu, Yonghwi Kwon, and Weihang Wang. An empirical study of bugs in webassembly compilers. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 42–54. IEEE, 2021.

28 Jeff Sauro and Jim Lewis. How to interpret a sus score, 2023. Accessed: 2025-05-17. URL: https://measuringu.com/interpret-sus-score/.

29 Snyk Limited. Deepcode ai ai code review - snyk.io. https://snyk.io/platform/deepcode-ai/. [Accessed 15-05-2025].

30 Jakub Swacha and Wiktor Przetacznik. In-Browser Implementation of a Gamification Rule Definition Language Interpreter. *Information*, 15(6), 2024. doi:10.3390/info15060310.

31 Rania Taleb, Sylvain Hallé, and Raphaël Khoury. Uncertainty in runtime verification: A survey. *Computer Science Review*, 50:100594, 2023.

32 Berik I Tuleuov and Ademi B Ospanova. *Beginning C++ Compilers*. Springer, 2024.

33 Melissa Warr, Nicole Jakubczyk Oster, and Roger Isaac. Implicit bias in large language models: Experimental proof and implications for education. *Journal of Research on Technology in Education*, pages 1–24, 2024.

34 Vithya Yogarajan, Gillian Dobbie, and Te Taka Keegan. Debiasing large language models: research opportunities. *Journal of the Royal Society of New Zealand*, 55(2):372–395, 2025.

35 George Zahariev. Grasp - JavaScript structural search, replace, and refactor — graspjs.com. `https://www.graspjs.com/`. [Accessed 15-05-2025].

36 Yixuan Zhang, Mugeng Liu, Haoyu Wang, Yun Ma, Gang Huang, and Xuanzhe Liu. Research on webassembly runtimes: A survey. *ACM Transactions on Software Engineering and Methodology*, 2024.