Browser-Sized Agents: Small Language Models and WebLLM for Tool-Augmented AI-Assisted Programming Education*

Bogdan Suvernev[1], Yuchen Wang[2,*], Xindi Tong[2] and Chee Wei Tan[2,*]

[1]City University of Hong Kong, Hong Kong SAR
[2]Nanyang Technological University, Singapore

Abstract

Programming courses are an early and demanding test bed for educational language models: they require code generation, code reading, multi-step reasoning across files, fast feedback under classroom latency, and an honest separation between what the model produced and what the student understood. We argue that small language models (SLMs), defined as open-weight models below ten billion parameters capable of running on consumer hardware, are well aligned with this setting, provided they are deployed as *tool-augmented* agents rather than monolithic oracles. Our core contribution is a tool-augmented SLM architecture for programming education that pairs a small, fine-tuned planner with a curated set of deterministic tools: a code executor, a unit-test runner, a documentation retriever, and a knowledge-tracing module. We report a preliminary 150M-parameter Python code-completion model trained from scratch as a candidate implementation for the architecture's completion component. We connect this design to three contemporary threads: the economic necessity of moving inference to the edge to eliminate per-token costs; the recent position that SLMs are architecturally preferable for structured agentic systems; and the pedagogical legibility of minimal model implementations. Finally, we outline a pilot evaluation and discuss open questions around fine-tuning data, tool calibration, and the hardware-capability divide that local inference introduces.

Keywords

Small language models, AI-assisted programming, Tool-augmented generation, Knowledge tracing, In-browser inference, Computing education

1. Introduction

The last three years of language-model research have been dominated by scale: larger pre-training corpora, larger parameter counts, and larger context windows. For education, the calculus is different. A classroom deployment lives under constraints that pull in the opposite direction, specifically the privacy of student work, predictable inference latency, instructor-side cost, and equitable access on modest hardware. The SLM4ED workshop has framed this tension precisely by defining a small language model (SLM) as an open-weight model below ten billion parameters that can be locally deployed or cheaply hosted, and by asking how such models can be made useful for learners.

Three contemporary threads converge on the same answer. The first is industrial compression and scale: moving inference to the edge eliminates the per-token costs that make cloud LLMs prohibitively expensive for large cohorts. Furthermore, Hassabis [1] notes that in iterative workflows, the speed advantage of small, local models often outweighs the slight capability gap compared to frontier models. The second thread is academic: Belcak et al. [2] argue that SLMs are *inherently more suitable* for agentic AI systems, where models are invoked repeatedly for structured sub-tasks rather than open-ended dialogue. The third thread is pedagogical legibility: minimal implementations like nanoGPT [3] and

SLM4ED'26: The 1st Workshop of Small Language Models for Education, June 28, 2026, Seoul, Republic of Korea
*Accepted for presentation at SLM4ED'26.
*Corresponding author.
bsuvernev2-c@my.cityu.edu.hk (B. Suvernev); yuchen011@e.ntu.edu.sg (Y. Wang); to0002di@e.ntu.edu.sg (X. Tong); cheewei.tan@ntu.edu.sg (C. W. Tan)
0009-0005-2803-6069 (B. Suvernev); 0000-0002-5183-1248 (Y. Wang); 0009-0007-1706-4472 (X. Tong); 0000-0002-6624-9752 (C. W. Tan)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

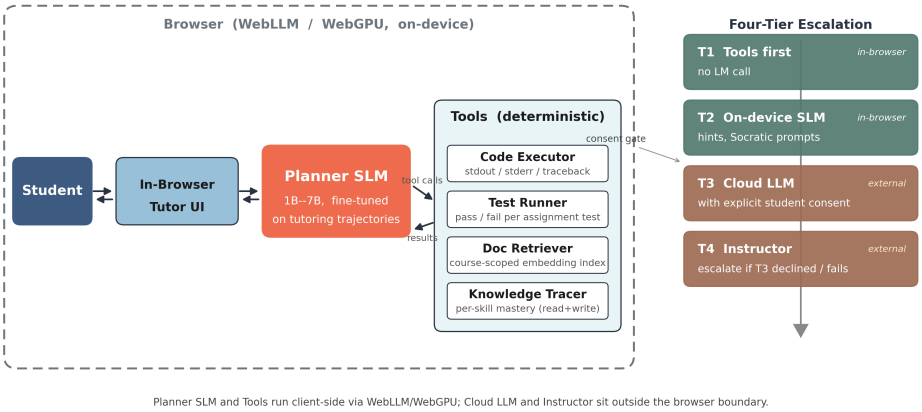

Figure 1: Tool-augmented SLM tutor for AI-assisted programming education. A small (1B–7B) planner model fine-tuned on tutoring trajectories orchestrates four deterministic tools (code executor, test runner, documentation retriever, knowledge tracer) inside the browser via WebLLM/WebGPU. Pedagogical control is governed by a four-tier escalation policy: *tools first*, *on-device SLM*, *cloud LLM with explicit student consent*, and *instructor escalation*. The first two tiers stay in-browser; the latter two cross the boundary only with consent.

microgpt [4] make the SLM itself a teachable object, small enough to modify and re-train on a single GPU.

These three threads are mutually reinforcing for AI-assisted programming education. The deployer wants a model whose inference is cheap and private; the system designer wants a model whose calls are short, structured, and routed through tools rather than asked to do everything in one prompt; and the educator wants a model whose internals can be taught. This paper makes the case that programming education is a highly compelling near-term application of SLMs, and our core contribution is sketching a tool-augmented SLM architecture (Figure 1) that operationalizes the three threads above. We connect the design to recent work on knowledge tracing for programming courses [5], which provides the pedagogical substrate for deciding what the student knows and what the system should say next, as well as to classroom-flipping approaches that use large language models to drive student-centric peer questioning [6]. We also connect it to in-browser inference engines such as WebLLM [7], which make it realistic to ship the entire tutor as a static web page that runs on the student's own GPU.

The rest of the paper traces the shift from cloud models to edge deployment (Section 2); argues why programming education fits the SLM definition (Section 3); presents two student-built WebLLM case studies (Section 4); develops the tool-augmented architecture together with its hybrid escalation policy (Section 5); and closes with a preliminary implementation of a ∼150M-parameter Python code-completion model trained from scratch (Section 7), alongside deployment notes, a pilot evaluation, and open questions (Sections 6–10).

2. Related Work: From Cloud Codex to On-Device Foundation Models

The integration of AI into programming education can be understood through a four-year trend in which the locus of deployment has steadily shifted from the cloud toward the edge. Rather than a strict taxonomy, this shift reflects an ongoing transfer of responsibility from remote servers to client devices.

Cloud-only LLM code generation (2021–2022). The starting point was the Codex paper [8], which established the paradigm that code generation occurs behind a remote API. In this model, the cloud handles all reasoning, but the client incurs network latency and per-token costs.

Thick-client IDE integration (2023–2024). The next step thickened the client. Systems like Copilot for Xcode [9] integrated cloud LLMs directly into the IDE, and subsequent work extended these copilots

from code generation toward software testing via context-based retrieval-augmented generation [10]. This made it clear that perceived quality depends heavily on IDE-side scaffolding and prompt construction, while exposing the limitations of cloud-only dependence: round-trip latency, billing, and privacy concerns.

Capable on-device foundation models (2025). The current shift involves foundation-quality models small enough to run on the developer's own device. Apple's FoundationModels framework [11], introduced at WWDC 2025, exposes a ∼3B-parameter on-device model optimized through KV-cache sharing and 2-bit quantization-aware training. This model is paired with a server-scale MoE model on Apple's Private Cloud Compute for spillover capacity. Three properties matter: the on-device model fits the SLM4ED sub-10B definition; the framework provides first-class *guided generation* and *tool calling*; and the server model is reserved for spillover rather than acting as the default. While frontier models continue to advance rapidly, this hybrid approach represents a practical, production-scale instantiation of local-first architecture.

Agentic SLMs as the default substrate. The most recent thread [2] generalizes this local-first approach into a research position. When language-model invocations are short, structured, and repeated, SLMs are not just sufficient but *architecturally preferable*, with frontier LLMs reserved for genuinely open-ended tasks. WebLLM [7] extends the same logic to the browser, removing the cloud entirely for compatible tasks.

Reading the trend. This trajectory describes the cloud relinquishing first the prompt format, then the model weights, and finally the agentic policy itself. AI-assisted programming education sits naturally at the edge of this trend. SLMs are not intended to replace frontier models; rather, they serve as the fast, local, and private orchestrators that make classroom deployment viable.

3. Why Programming Education Fits the SLM Definition

The workshop defines an SLM as an open-weight model below ten billion parameters that can be locally deployed or cheaply hosted. For programming education, three properties of this definition matter. First, a programming course requires a *bounded task family*, such as explaining compiler errors or suggesting test cases. Belcak et al. [2] note that when a model is invoked repeatedly for narrow, specialized roles, the marginal value of frontier generality drops while its cost does not. Second, pedagogical interaction requires *tight feedback loops*. Hassabis [1] observes that a small model running significantly faster than a frontier model often dominates in iterative workflows, favoring architectures that decompose questions into short, fast calls.

Finally, local inference introduces a complex dynamic regarding *privacy and equity*. Running models entirely in the student's browser (Section 6) avoids the regulatory complications of sending learner data to a third-party cloud and removes the per-token cost barrier that prevents under-resourced schools from offering AI-assisted instruction. However, this approach is double-edged: while it eliminates subscription costs, it introduces a hardware-capability divide. Students with modern, high-end laptops will experience fast, seamless generation, whereas students on older hardware may face severe latency or memory exhaustion. Any deployment of SLMs in education must acknowledge and design around this new hardware gap. Ultimately, knowledge tracing for programming courses [5] already combines generative AI with adaptive analytics under privacy-preserving goals, making local SLMs a structurally appropriate choice for the task.

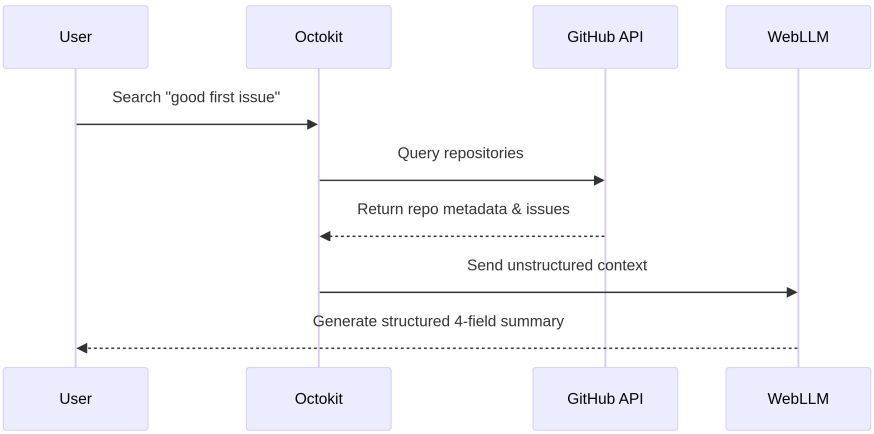

Figure 2: SearchOSR Workflow: Octokit retrieves repository metadata and issues, which are summarized on-device by WebLLM.

4. In-Browser Agentic Workflows: Two WebLLM Case Studies

To demonstrate the feasibility of the Phase 3/4 transition identified in Section 2, we present two student-built case studies deploying agentic workflows entirely within the browser using WebLLM [7]. Both projects tackle the constraints of client-side inference, specifically latency, cross-origin isolation, and structured generation. These constraints are central to building a tool-augmented programming tutor.

4.1. The Unified AI Gateway (UAG)

The Unified AI Gateway (UAG) is a Chromium Manifest V3 extension designed to abstract the choice between local and remote LLMs for web applications. The extension provides a unified interface where a user can select either an in-browser model (e.g., Llama-3 or Phi-3-mini running via WebLLM's IndexedDB cache and WebGPU) or a remote API endpoint (e.g., an Ollama instance).

A key challenge was browser CORS restrictions blocking direct extension-to-Ollama calls. Two solutions were compared: (1) a Flask middleware proxy that adds the appropriate headers, providing portability and centralized control at the cost of an additional hop; and (2) configuring the backend (Ollama) to whitelist the extension's origin via a permissive `OLLAMA_ORIGINS` setting, which has lower overhead but a wider attack surface. On a consumer laptop (Lenovo Yoga 7, Intel i7-1165G7, Iris Xe), the remote API completes requests in 10 to 58 seconds, while in-browser WebLLM requires roughly 180 seconds per trial. This highlights a critical reality: edge inference is not universally faster than a foundation service. Furthermore, downloading gigabytes of model weights to the browser cache is a significant initial barrier, and low-end devices like Chromebooks lack the memory to run an 8B model or handle multiple WebLLM-enabled tabs simultaneously. Interactive pedagogical latency still depends entirely on a capable client-side GPU.

4.2. SearchOSR: Structured Summarization

The second case study, SearchOSR, combines the GitHub Search and Repository APIs (via Octokit) with WebLLM to help beginner programmers find and understand open-source issues labeled `good first issue`.

SearchOSR exposes a critical challenge for agentic SLMs: structured generation. The system requires the SLM to read unstructured repository metadata and issue threads and emit a strict four-field summary (repository description, issue context, prerequisites, likely challenges). Unstructured prompts produced verbose, unparseable outputs; explicit field enumeration plus low temperatures (0.2 for extraction vs. 0.7 for creative generation) improved reliability, though raising `max_tokens` for completeness severely impacted latency. This manual prompt engineering is precisely the kind of brittle structured-output

problem addressed by XGrammar [12] and extended by MLC's May 2026 XGrammar 2 release [13]: dynamic tag dispatch and constrained decoding allow small models running within WebLLM to produce schema-compliant summaries and tool-consumable outputs with near-zero overhead, thereby making browser-side WebLLM workflows more robust as tool-augmented agents.

5. A Tool-Augmented SLM Architecture for AI-Assisted Programming

Two features distinguish a useful programming tutor from a chat box that happens to run on a small model. The first is *tools*: a code executor, a test runner, a documentation index, and a knowledge-tracing module recording what the student has shown they understand. The second is a *planner* that decides which tools to call and in what order. Belcak et al. [2] explicitly argue that the planner is the most numerous and repetitive call in the system, and therefore benefits most from being a small fine-tuned model rather than a large general one.

Components and Tool Curation. We propose six components for an introductory programming course: (i) a **Planner SLM**, a small (1B to 7B) instruction-tuned model fine-tuned on tutoring trajectories that translates a student message into a sequence of tool calls; (ii) a **Completion SLM**, a very small local model responsible for inline code autocompletion in the student's editor (detailed in Section 7); (iii) a **Code executor**, a sandboxed interpreter returning stdout/stderr/tracebacks as the source of ground truth for code execution; (iv) a **Test runner** wrapping the student's assignment test suite for pass/fail reports that drive hint decisions; (v) a **Documentation retriever**, a small embedding model and course-scoped index over lecture notes and language references; and (vi) a **Knowledge-tracing module** in the style of [5], maintaining per-skill mastery estimates that the planner reads to decide hint level and writes to as new evidence arrives.

Crucially, the list of tools must be curated tightly. As model capability drops, the ability to invoke tools correctly also drops. This degradation is often exacerbated by aggressive quantization techniques required to fit models into browser memory, rather than just the raw parameter count. Therefore, we must choose only a few high-level tools, switch as much logic as possible into deterministic code, and train the SLM precisely to this constrained set. In WebLLM, XGrammar [12] and XGrammar 2 [13] can constrain the Planner SLM to emit schema-valid agent tool calls rather than free-form text.

Why tool augmentation is pedagogically necessary. A monolithic LLM tutor that "just answers" deprives the student of the formative feedback that comes from running their own code. A tool-augmented architecture forces the model to externalize those steps. A hint becomes "the test runner reports that `test_edge_cases` fails on input `n=1`; what do you think your function returns when the list is empty?" rather than "the bug is on line 7." The planner produces explanations grounded in tool outputs, which the student can re-run and inspect. This distinction is critical: it shifts the system from an oracle that merely provides answers to a formative tool that models the debugging process.

5.1. Three Architectures and a Hybrid Fourth

The deployment trends of Section 2 correspond to three architectures and a hybrid. **(A) Cloud-only LLM** (Codex [8], Copilot for Xcode [9]) offers frontier capability but suffers per-call cost, network-bound latency, and a privacy posture dependent on a contractual relationship with the provider. **(B) On-device monolithic SLM** trades capability for cost, latency, and privacy. It is rational for narrow tasks but fails on multi-step reasoning, which is exactly the regime where such models tend to produce confident but incorrect answers. **(C) On-device tool-augmented SLM** reduces the SLM to a planner; deterministic tools carry the load a monolithic model would otherwise handle implicitly. **(D) Hybrid**: Apple's production design [11] layers an on-device SLM (handling structured outputs and Swift tool calls) with a server-scale MoE on Private Cloud Compute reserved for spillover.

For an AI programming tutor, we propose a four-tier escalation routed by the planner: *tools first* (deterministic tool answers without invoking any LM); *on-device SLM* (for hints, explanations, Socratic re-phrasings); *cloud LLM with student consent* (for questions exceeding planner competence, with prompt and response logged for instructor review); and *instructor escalation* (if consent is refused or the cloud LLM fails a quality check). This explicit consent mechanism ensures the hybrid model remains pedagogically defensible. Instead of a transparent fallback that silently masks the local model's failure, cloud escalation becomes a deliberate, logged, and reviewable act.

Implications. We view this hybrid approach as an appropriate transition strategy for the next couple of years, prioritizing latency and cost savings while edge hardware catches up. It inverts the cost curve of the cloud-only baseline because the most expensive part is called least often. Evaluation must be reorganized: a leaderboard score on a generic benchmark says little about whether the right call was routed to the right tier. Most consequentially, the locus of pedagogical control shifts from the model to the *policy* that orchestrates it, including fine-tuning data, tool APIs, escalation rules, and consent UX. Each of these elements can be inspected, modified, and studied by the educational community.

6. In-Browser Deployment

WebLLM [7] demonstrates that 0.5B to 8B models can be served entirely client-side via WebGPU. For an SLM tutor, this addresses three concerns simultaneously. First, *distribution* becomes a static site with weights cached in IndexedDB and tools running in JavaScript or WebAssembly, requiring no API keys or per-call billing. Second, *privacy* becomes the default, as student code, prompts, and transcripts never leave the device. Finally, the system gains a degree of *transparency*. While true explainability remains an open research problem, a curious student can at least open the browser's developer tools to observe the prompts the planner emits and the tool responses it receives. When this approach is combined with readable model implementations like nanoGPT and microgpt [3, 4], the entire software stack becomes more accessible to inspection than a proprietary cloud service.

Narrow tasks versus planning. The architecture above does not require every component to be a model with billions of parameters. Some interactions in a programming course have a much narrower scope. For example, completing a started line, predicting the next few tokens within a function, or continuing a code snippet based on the context of the surrounding file are frequent, latency-sensitive tasks tightly coupled to the student's editor. To explore this lower end of the design space, we trained a preliminary ∼150M-parameter Python code-completion model as a candidate implementation of the completion component in the proposed tutor.

7. Preliminary Implementation: LightLM, a ∼150M Python Autocomplete SLM

As a candidate completion component (Section 5), we trained *LightLM*, a ∼150M-parameter SLM for Python code completion.[1] It is designed strictly for short continuations and fragment restoration near the student's cursor; pedagogically sensitive decisions (hinting, testing, escalation) remain with the planner. The model has completed English pretraining, code pretraining, and a compact supervised fine-tuning (SFT) phase. We report architectural choices and pretraining zero-shot results (Table 1).

Architecture and English pretraining. To use 150M parameters efficiently, we compared three variants: (i) attention-focused; (ii) FFN-focused; and (iii) a dense (soft) mixture-of-experts model [14]. All share the SmolLM2-360M tokenizer [15] and were trained for 28B tokens on Cosmopedia v2 [16].

[1] Pretraining code: https://github.com/virg1n/LightLM; fine-tuning code: https://github.com/virg1n/finetuning-lightlm; model weights: https://huggingface.co/Virg1n/LightLM.

The three variants reached performance comparable to MobileLLM-125M [17] on ARC-C [18] (Table 1). We proceeded with the FFN-focused variant.

Table 1
Zero-shot ARC-C [18] accuracy (%) of LightLM variants vs MobileLLM [17].

Model	Parameters	ARC-C
MobileLLM-125M	125M	27.1
LightLM (Attn)	146M	25.1
LightLM (FFN)	146M	27.2
LightLM (Soft-MoE)	144M	26.3

Code pretraining and FIM. We continued pretraining on 80% code, 15% technical English, and 5% Cosmopedia replay (to retain fluency for parsing inline comments). Python was extracted from StarCoderData [19, 20], CodeParrot-clean [21], and GitHub Jupyter notebooks [22]; non-Python code from StarCoderData/The Stack [20, 23]. Because autocomplete conditions on context both before and after the cursor, we adopted the Fill-in-the-Middle objective [24, 25], ramping to a 60%/40% FIM/L2R mix over the first 11B tokens. Context length was increased from 1,024 to 2,048 tokens.

Table 2
HumanEval pass@1 accuracy (%) of LightLM after supervised fine-tuning compared with small code-oriented baselines.

Model	Parameters	HumanEval pass@1
GPT-Neo	125M	0.75
AlphaCode	89M	4.30
CodeParrot	110M	3.80
PyCodeGPT	110M	8.33
LightLM (ours)	**146M**	**10.97**
CodeGen-Mono	350M	12.76

Supervised fine-tuning. We are conducting a compact ($\sim$2B-token) SFT phase to align the model with instructional queries and docstring-driven completion. The mix is 70% instruction-style and 30% raw completion. The instruction portion combines OpenCodeInstruct [26] and Magicoder OSS-Instruct [27]; Evol-Instruct-Code [28]; Code-Feedback [29] and CodeUltraFeedback [30] for multi-step dialogues; SelfCodeAlign [31]; and a 5% slice of synthetic HumanEval-format pairs [8] (from non-overlapping seeds [32]). Context length was extended to 4,096 tokens. After fine-tuning, LightLM reaches 10.97% pass@1 on HumanEval (Table 2), which should be interpreted as a preliminary functional-correctness sanity check rather than as the primary educational evaluation of the system. Nevertheless, it places LightLM above several previously reported 100M–150M code-oriented baselines and below the larger 350M CodeGen-Mono model.

8. Pilot Evaluation Plan

We outline a pilot comparing three conditions in an introductory programming course: a **Baseline** (existing course copilot with a frontier cloud LLM); **SLM-direct** (the same UI but a 7B SLM with no tool augmentation); and **SLM-tool-augmented** (the architecture of Section 5). The primary outcome is the per-skill mastery trajectory from the knowledge-tracing module, evaluated against end-of-week quizzes in the spirit of [5]. Secondary measures include median time-to-resolution, frequency of off-syllabus answers, and self-reported student trust. The hypothesis is that **SLM-tool-augmented** is non-inferior

to **Baseline** on mastery and superior on latency, privacy, and inspectability, whereas **SLM-direct** on its own is not.

9. Limitations

We acknowledge several limitations in this work. First, the proposed architecture is presented as a design and preliminary implementation; we have not yet conducted a controlled classroom experiment to measure long-term learning gains or skill transfer. Second, while the 150M-parameter completion model demonstrates the feasibility of training small, targeted components, its impact on student reliance and debugging confidence remains to be evaluated. Finally, we have not yet benchmarked the full tool-augmented planner against existing knowledge-tracing datasets or standard code generation benchmarks, which will be necessary to quantify the exact capability trade-offs of moving from cloud LLMs to edge SLMs.

10. Questions for the Community

We close with four questions well-posed for the SLM4ED community:

1. **Fine-tuning corpora:** What is the minimum viable set of trajectories needed to train a tool-augmented planner, and how much can be synthetically generated by a frontier teacher model?
2. **Tool calibration:** How do we train the planner to reliably invoke tools rather than answering directly, optimizing for pedagogical evidence rather than mere task accuracy?
3. **Curriculum evaluation:** How do we evaluate SLM tutors on whether students learned faster or at lower institutional cost [5], rather than on generic coding benchmarks?
4. **Autonomous research workflows:** As lightweight "autoresearch" agents begin to automate parts of the research and coding loop [33], how should an SLM tutor scaffold these workflows so that students still build foundational understanding rather than outsourcing it?

11. Conclusion

Small language models, defined as open-weight and locally deployable models below ten billion parameters, are not merely a fallback for settings that cannot afford a frontier model. Rather, they are an architecturally appropriate substrate for AI-assisted programming education, provided they are deployed as tool-augmented planners rather than monolithic oracles. Pedagogically legible implementations [3, 4], the agentic-systems argument [2], edge-side distillation [1], production on-device foundation models [11], in-browser engines [7], and programming-course knowledge tracing [5] jointly describe a deployable design. Our two WebLLM case studies demonstrate that the in-browser component is now practical, and our preliminary 150M-parameter Python completion model is a first step toward instantiating the completion component of the proposed architecture. The trend from cloud generation [8] through IDE copilots [9] to agentic SLMs culminates, for education, in a four-tier hybrid where most inference is local, tool calls carry most of the load, and cloud escalation is explicit and consented. We look forward to discussing with the SLM4ED community whether this hybrid approach can effectively support student learning.

Acknowledgments

This research is supported by the Singapore Ministry of Education Academic Research Fund (MOE-T2EP20224-0009) and NTU startup fund.

Declaration on Generative AI

The author(s) used generative AI tools (ChatGPT, Grammarly) for grammar checking and language polishing of the manuscript. No AI system was used to generate experimental results or model outputs reported as findings in this paper.

References

[1] D. Hassabis, DeepMind founder visits Y Combinator: On AGI, agents, and distillation into small models, Y Combinator podcast, summarized by PANews, 2026. URL: https://www.panewslab.com/en/articles/019dde05-2a76-7628-955f-3976151af4d5.

[2] P. Belcak, G. Heinrich, S. Diao, Y. Fu, X. Dong, S. Muralidharan, Y. C. Lin, P. Molchanov, Small language models are the future of agentic AI, 2025. URL: https://arxiv.org/abs/2506.02153. arXiv:2506.02153.

[3] A. Karpathy, nanoGPT: The simplest, fastest repository for training/finetuning medium-sized GPTs, GitHub repository, 2022. URL: https://github.com/karpathy/nanoGPT.

[4] A. Karpathy, microgpt: A minimal dependency-free GPT in pure Python, Personal blog, 2026. URL: https://karpathy.github.io/2026/02/12/microgpt/.

[5] Y. Wang, P.-D. Yu, C. W. Tan, Future-proofing programmers: Optimal knowledge tracing for artificial intelligence-assisted personalized education [special issue on artificial intelligence for education: A signal processing perspective], IEEE Signal Processing Magazine 43 (2026) 69–82. doi:10.1109/MSP.2025.3609896.

[6] C. W. Tan, Large language model-driven classroom flipping: Empowering student-centric peer questioning with flipped interaction, 2023. URL: https://arxiv.org/abs/2311.14708. arXiv:2311.14708.

[7] C. F. Ruan, Y. Qin, X. Zhou, R. Lai, H. Jin, Y. Dong, B. Hou, M.-S. Yu, Y. Zhai, S. Agarwal, H. Cao, S. Feng, T. Chen, WebLLM: A high-performance in-browser LLM inference engine, 2024. URL: https://arxiv.org/abs/2412.15803. arXiv:2412.15803.

[8] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., Evaluating large language models trained on code, 2021. URL: https://arxiv.org/abs/2107.03374. arXiv:2107.03374.

[9] C. W. Tan, S. Guo, M. F. Wong, C. N. Hang, Copilot for Xcode: Exploring AI-assisted programming by prompting cloud-based large language models, 2023. URL: https://arxiv.org/abs/2307.14349. arXiv:2307.14349.

[10] Y. Wang, S. Guo, C. W. Tan, From code generation to software testing: AI copilot with context-based retrieval-augmented generation, IEEE Software 42 (2025) 34–42. doi:10.1109/MS.2025.3549628.

[11] Apple, Apple intelligence foundation language models: Tech report 2025, arXiv preprint arXiv:2507.13575 (2025).

[12] Y. Dong, C. F. Ruan, Y. Cai, R. Lai, Z. Xu, Y. Zhao, T. Chen, XGrammar: Flexible and efficient structured generation engine for large language models, 2025. URL: https://arxiv.org/abs/2411.15100. arXiv:2411.15100.

[13] L. Li, Y. Dong, G. Wang, Z. Xu, A. Jiang, T. Chen, XGrammar-2: Efficient dynamic structured generation engine for agentic LLMs, 2026. URL: https://arxiv.org/abs/2601.04426. arXiv:2601.04426.

[14] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, J. Dean, Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, in: International Conference on Learning Representations, 2017. URL: https://arxiv.org/abs/1701.06538.

[15] L. B. Allal, A. Lozhkov, E. Bakouch, G. M. Blázquez, G. Penedo, L. Tunstall, A. Marafioti, H. Kydlíček, A. P. Lajarín, V. Srivastav, J. Lochner, C. Fahlgren, X.-S. Nguyen, C. Fourrier, B. Burtenshaw, H. Larcher, H. Zhao, C. Zakka, M. Morlon, C. Raffel, L. von Werra, T. Wolf, SmolLM2: When smol goes big – data-centric training of a small language model, 2025. URL: https://arxiv.org/abs/2502.02737. arXiv:2502.02737.

[16] Hugging Face Smol Models Research, SmolLM-corpus, https://huggingface.co/datasets/HuggingFaceTB/smollm-corpus, 2024. Hugging Face dataset card, subset "cosmopedia-v2", accessed 2026-05-14.

[17] Z. Liu, C. Zhao, F. Iandola, C. Lai, Y. Tian, I. Fedorov, Y. Xiong, E. Chang, Y. Shi, R. Krishnamoorthi, L. Lai, V. Chandra, MobileLLM: Optimizing sub-billion parameter language models for on-device use cases, arXiv preprint arXiv:2402.14905 (2024). URL: https://arxiv.org/abs/2402.14905.

[18] P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, O. Tafjord, Think you have solved question answering? try ARC, the AI2 reasoning challenge, arXiv preprint arXiv:1803.05457 (2018). URL: https://arxiv.org/abs/1803.05457.

[19] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, et al., Starcoder: may the source be with you!, 2023. URL: https://arxiv.org/abs/2305.06161. arXiv:2305.06161.

[20] BigCode Project, Starcoderdata, https://huggingface.co/datasets/bigcode/starcoderdata, 2023. Hugging Face dataset card, accessed 2026-05-14.

[21] L. von Werra, Training codeparrot from scratch, https://huggingface.co/blog/codeparrot, 2021. Hugging Face blog post.

[22] Hugging Face CodeParrot Team, Github jupyter dataset, https://huggingface.co/datasets/codeparrot/github-jupyter-parsed, 2022. Hugging Face dataset card, accessed 2026-05-14.

[23] D. Kocetkov, R. Li, L. Ben Allal, J. Li, C. Mou, C. Muñoz Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, H. de Vries, The stack: 3 TB of permissively licensed source code, arXiv preprint arXiv:2211.15533 (2022). URL: https://arxiv.org/abs/2211.15533.

[24] M. Bavarian, H. Jun, N. Tezak, J. Schulman, C. McLeavey, J. Tworek, M. Chen, Efficient training of language models to fill in the middle, arXiv preprint arXiv:2207.14255 (2022). URL: https://arxiv.org/abs/2207.14255.

[25] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, M. Lewis, Incoder: A generative model for code infilling and synthesis, in: International Conference on Learning Representations, 2023. URL: https://arxiv.org/abs/2204.05999.

[26] W. U. Ahmad, A. Ficek, M. Samadi, J. Ross, et al., Opencodeinstruct: A large-scale instruction tuning dataset for code LLMs, 2025. URL: https://arxiv.org/abs/2504.04030. arXiv:2504.04030.

[27] Y. Wei, Z. Wang, J. Liu, Y. Ding, L. Zhang, Magicoder: Empowering code generation with OSS-instruct, in: Proceedings of the 41st International Conference on Machine Learning (ICML), 2024.

[28] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, D. Jiang, WizardCoder: Empowering code large language models with evol-instruct, in: The Twelfth International Conference on Learning Representations (ICLR), 2024.

[29] T. Zheng, G. Zhang, T. Shen, X. Liu, B. Y. Lin, J. Fu, W. Chen, X. Yue, OpenCodeInterpreter: Integrating code generation with execution and refinement, arXiv preprint arXiv:2402.14658 (2025). URL: https://arxiv.org/abs/2402.14658.

[30] M. Weyssow, A. Kamanda, X. Zhou, H. Sahraoui, CodeUltraFeedback: An LLM-as-a-judge dataset for aligning large language models to coding preferences, 2024. URL: https://arxiv.org/abs/2403.09032. arXiv:2403.09032.

[31] Y. Wei, F. Cassano, J. Liu, Y. Ding, N. Jain, Z. Mueller, H. de Vries, L. Zhang, SelfCodeAlign: Self-alignment for code generation, in: Advances in Neural Information Processing Systems (NeurIPS), volume 37, 2024.

[32] M. Riddell, A. Ni, A. Cohan, Quantifying contamination in evaluating code generation capabilities of language models, in: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2024, pp. 14116–14137. URL: https://arxiv.org/abs/2403.04811.

[33] A. Karpathy, autoresearch, GitHub repository, 2026. URL: https://github.com/karpathy/autoresearch.