Engineering a Scalable Laboratory Infrastructure for Assembly Language Scaffolding: Design and Deployment of a Locally Optimized GenAI Assistant for the CODE-2 Educational Architecture

[Blinded for Double-Blind Peer Review]

Running title: GenAI Assembly Scaffolding: CODE-2 Lab

Abstract

The transition from high-level programming to assembly language constitutes a well-documented pedagogical bottleneck in computer engineering curricula, particularly in large-cohort laboratory settings where individualized scaffolding cannot scale. This paper presents the design, implementation, and technical evaluation of a locally deployable generative AI assistant engineered specifically for the CODE-2 educational processor architecture. The system is intended as laboratory infrastructure, not as a replacement for human instruction; its primary contribution is enabling scalable, privacy-preserving syntax scaffolding without dependency on cloud services or internet connectivity. A synthetic task bank of 50,000 instruction pairs was procedurally generated to cover the full CODE-2 curriculum. Three fine-tuning strategies were compared on a consumer GPU: Prompt Tuning, Low-Rank Adaptation (LoRA), and Full Fine-Tuning of a T5-Small encoder-decoder model. Full Fine-Tuning achieved 94.10% Exact Match on the held-out evaluation set, demonstrating that rigid assembly syntax requires full parameter adaptation. Post-training INT8 quantization via ONNX Runtime reduced inference latency by 69% (from 1,689 ms to 526 ms) on standard laboratory hardware (Intel i5, 8 GB RAM), with a precision loss below 1%. The resulting system operates entirely offline, precluding data exfiltration by design. The system is integrated into laboratory workflows as a supervised scaffolding tool, requiring mandatory emulator-based verification of all AI-generated code. Pedagogical implications are discussed as plausible benefits; no controlled learning-gains study is reported. The work demonstrates a replicable pipeline for building domain-specific language model infrastructure tailored to CPU-only educational environments.

Keywords: engineering education; assembly programming; fine-tuning; quantization; laboratory infrastructure

1 Introduction

1.1 Curricular Imperatives and the Abstraction Gap

Mastery of Computer Architecture is a foundational requirement in the training of computer and electronic systems engineers. Understanding the hardware-software interface is not an auxiliary skill but a primary competency for optimizing performance-critical systems and reasoning about hardware-level security. The pedagogical transition from high-level programming paradigms (Python, Java, C++) to assembly language constitutes one of the most significant hurdles in student progression [1].

In high-level environments, students rely on compilers that automate memory management, handle stack frames, and resolve variable scopes. In assembly, these abstractions are removed. Students must internalize a restricted register file, explicit addressing modes, and the rigid fetch-decode-execute cycle. This paradigm shift demands the construction of what Ben-Ari calls "viable mental models" [1]: without them, students are unable to initiate even simple translations of algorithmic intent into machine instructions.

1.2 The Blank Page Syndrome and Cognitive Load

In laboratory settings, this burden frequently manifests as the "blank page syndrome." Engineering students can often articulate the logic of a problem (e.g., finding the maximum of an array) yet cannot begin translating that logic into assembly. Managing both algorithmic logic and syntactic constraints simultaneously risks exceeding working memory capacity, consistent with Cognitive Load Theory [2].

Implementing a simple conditional loop in a pedagogical ISA such as CODE-2 requires initializing counters, managing memory pointers, manipulating comparison flags, and orchestrating branch labels—all before a single useful instruction can be written. The effort of recalling exact opcodes and operand ordering consumes cognitive resources that should be directed toward structural understanding. This bottleneck has been identified as a contributor to student disengagement in architecture courses [3].

1.3 The Scalability Problem in Laboratory Instruction

Contemporary engineering programs often enroll more than 100 students per laboratory session. The resulting instructor-to-student ratio precludes immediate, personalized scaffolding. Scaffolding—providing temporary, calibrated support to help a learner operate within the Zone of Proximal Development [4]—is most effective when delivered at the precise moment of need.

When a student is blocked by a syntax question, they must wait in a queue for human assistance. This downtime erodes self-efficacy, encourages disengagement, and disrupts the laboratory workflow. Existing educational software for CODE-2 (emulators, debuggers) allows

post-hoc verification but offers no generative assistance. This creates a productive gap that current tools do not address.

1.4 Contribution: A Systems-Engineering Approach

To address this infrastructure gap, we present a locally deployable Generative AI (GenAI) assistant specifically engineered for the CODE-2 educational environment. The primary contribution is *infrastructure design*, not model novelty. The technical choices—model size, quantization strategy, offline-only operation, CPU inference—are driven by the constraints of university laboratory environments (CPU-only hardware, restricted internet access, GDPR obligations) rather than by a desire to advance state-of-the-art language modeling.

Unlike general-purpose cloud-based Large Language Models [5], which introduce privacy risks, incur recurring API costs, and may hallucinate instructions from irrelevant commercial ISAs, our system is designed as a sovereign, hardware-aware laboratory tool. By fine-tuning and quantizing a T5-Small model [6], we demonstrate that effective syntax-level scaffolding can be delivered on standard university hardware without internet dependency.

The specific contributions of this work are:

1. A procedural data generation pipeline producing a 50,000-pair synthetic task bank covering the complete CODE-2 curriculum;

2. A comparative evaluation of Prompt Tuning, LoRA, and Full Fine-Tuning for rigid assembly ISA adaptation, providing quantitative evidence that lightweight PEFT methods are insufficient for this class of formal language;

3. An INT8 post-training quantization pipeline enabling CPU-only deployment with a 69% reduction in inference latency and less than 1% precision loss;

4. A robustness stress test on the deployable INT8 pipeline ($n = 2{,}000$), treating input noise—typographical errors, truncated intents, and Spanish–English code–mixing—as an operational constraint for classroom deployment. The test quantifies both the system's reliable regimes and its failure modes under matched preprocessing and decoding, complementing the primary Exact Match evaluation (Section 6.4).

5. A supervised integration protocol framing the assistant as a scaffolding tool subject to mandatory emulator-based verification.

1.5 Related Work and Positioning

Engineering-education tooling for assembly laboratories has historically emphasized *emulation*, *visualization*, and post-hoc verification: students write code, run it in an emulator, and inspect datapath or memory-state effects to diagnose errors. Such tools are essential, but they do not

provide generative, step-by-step scaffolding at the moment a learner is blocked by a syntactic recall problem. In contrast, recent work on AI-assisted programming education highlights the need to frame LLM output as fallible guidance that must be audited by learners [7]. Our contribution is therefore positioned as laboratory *infrastructure*: an offline, CPU-deployable assistant that complements emulator/visualization workflows rather than replacing them, and that is integrated with mandatory verification and feedback loops aligned with scaffolding practices in architecture education [3]. The remainder of this paper is organized as follows: Section 2 introduces the CODE-2 context, Sections 3–6 describe the engineering pipeline and technical evaluation, Section 7 details laboratory integration, and Sections 8–9 discuss pedagogical implications, limitations, and transferability.

2 Educational Context: The CODE-2 Architecture

2.1 Architectural Transparency as a Pedagogical Design Principle

CODE-2 is a 16-bit RISC processor designed specifically for instructional transparency, as described by Prieto et al. [8]. Its deliberate simplicity exposes students to the full datapath without the complexity of commercial ISAs (x86, ARM). CODE-2 is used in Computer Architecture laboratory courses as the primary environment for learning assembly programming.

2.1.1 Register File and Memory Model

CODE-2 provides a 16-bit unified address space (65,536 addressable word locations). The processor includes:

- **General-Purpose Registers (R0–R9):** Ten registers for data manipulation and temporary storage.
- **Accumulator (RA):** Implicit destination for arithmetic operations, requiring explicit data-movement instructions.
- **Flags Register (RF):** Status bits for Zero (Z), Sign (S), Carry (C), and Overflow (O); flag management is a primary learning outcome.

2.1.2 ISA Rigidity and Addressing Modes

The instruction set supports Immediate, Direct, and Indirect addressing. The assembler enforces a strict `OPCODE DEST, SRC1, SRC2` grammar. This rigidity is pedagogically intentional: it demands engineering precision. For a student uncertain of an exact mnemonic or operand order, however, this rigidity becomes an obstacle. Critically, CODE-2 is a proprietary academic ISA with no online community resources (no Stack Overflow threads, no tutorials); students depend entirely on course materials and instructor availability.

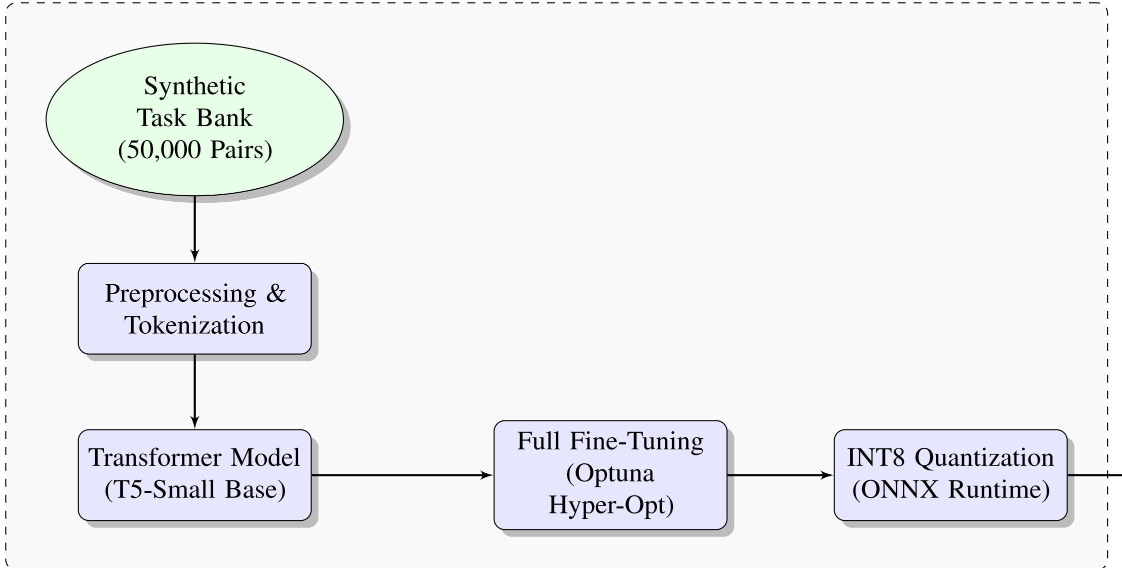
2.2 Data Sovereignty and GDPR in Educational AI

Engineering education must navigate the ethical and legal implications of AI tool deployment in university settings [4]. Cloud-based AI services introduce risks related to GDPR compliance (student queries may constitute personal data), API cost dependency, and pedagogical integrity (unrestricted access to solutions). Our architecture prevents data exfiltration by design: all inference occurs locally, no network I/O is initiated during operation, and student interaction logs remain on the institution's hardware. This constitutes an architectural privacy guarantee rather than a policy promise. The avoidance of cloud dependencies also eliminates the "hidden technical debt" associated with reliance on external APIs [9].

3 System Architecture and Implementation

We adopt a systems engineering pipeline that separates a computationally intensive offline training phase from a lightweight, CPU-only online deployment phase. The architecture (Figure 1) ensures the final tool runs on standard laboratory workstations without internet access.

Offline Engineering Pipeline (GPU-Enabled)



Online Classroom Workflow (Local CPU Only)

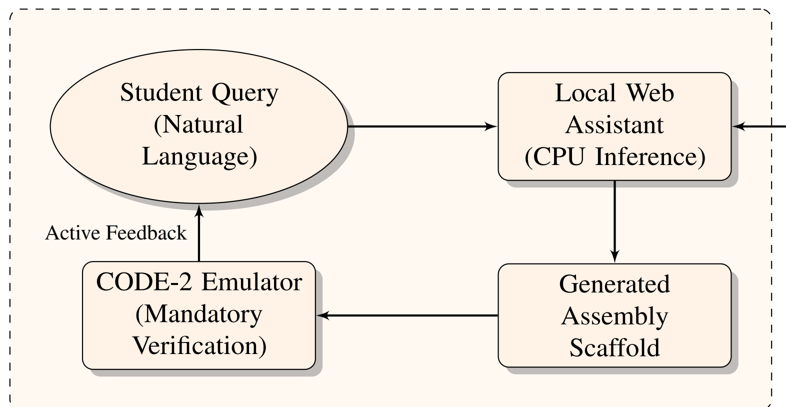


Figure 1: System architecture illustrating the two-phase pipeline. The offline phase (upper, grey) performs dataset generation, model training, and INT8 quantization on a GPU workstation. The online phase (lower, orange) runs entirely on standard laboratory PCs without internet connectivity; the CODE-2 emulator provides mandatory verification of all AI-generated scaffolds.

3.1 Custom Tokenization for Assembly Syntax

Standard NLP subword tokenizers (e.g., BERT's WordPiece) are optimized for natural language and fragment assembly mnemonics unpredictably (e.g., `ADDS` $\rightarrow$ `ADD` + `S`). To mitigate this, we trained a custom SentencePiece tokenizer [10] on the CODE-2 corpus:

- **Vocabulary size:** 32,000 tokens, balancing expressivity and model footprint.
- **Special tokens:** Structural delimiters (commas, brackets) are treated as atomic tokens to preserve operand boundaries.
- **Effect:** Custom tokenization reduced mean sequence length by 15% relative to the base T5 tokenizer, directly improving inference throughput.

3.2 Inference Engine and Deployment Stack

University laboratory workstations are typically mid-range machines (Intel i5 or equivalent, 8 GB RAM, integrated graphics). To ensure smooth CPU-only operation, we use ONNX Runtime [11] as the inference engine:

- **Model export:** The fine-tuned PyTorch model is exported to the ONNX graph format, decoupling training from inference infrastructure.
- **Execution provider:** Configured for the CPU Execution Provider with AVX2 instruction set extensions.
- **Interface:** A lightweight Flask-based web application runs on `localhost`; no data leaves the student's workstation. The assistant is therefore not a cloud service but a local process, analogous to a debugger.

4 Dataset Engineering: The Synthetic Task Bank

4.1 Procedural Data Generation

Because CODE-2 is a proprietary educational ISA, no pre-existing corpora or benchmarks are available. To overcome this cold-start problem, we engineered a synthetic task bank of 50,000 unique natural-language-to-assembly instruction pairs via a Python-based procedural generator.

The generator instantiates 200 pedagogical patterns, each corresponding to a specific curricular outcome. Patterns parameterize register names (R0–R9), memory addresses (0x0000–0xFFFF), and immediate values to produce diverse surface forms from a fixed underlying schema. For example, the "pointer dereference" pattern generates distinct pairs for every combination of source and destination register:

```
; NL: Load the value at the address stored in R1 into R2
  LD R2, [R1]
```

Listing 1: Example of synthetic pattern instantiation (pointer dereference).

4.2 Curriculum Coverage, Data Splits, and Integrity

The dataset was balanced across four complexity levels mirroring the course syllabus:

- **Level 1 — Atomic instructions (20%):** Single loads and moves (`LD`, `ST`, `MOV`).
- **Level 2 — Memory interaction (30%):** Indirect addressing and pointer arithmetic; critical for the hardware-software boundary.
- **Level 3 — ALU and flag logic (30%):** Comparison (`CMP`) and conditional branches (`JZ`, `JN`).
- **Level 4 — Control structures (20%):** Multi-instruction sequences for loops and conditionals, requiring label management.

The 50,000 pairs were divided 80/20 into training (40,000) and held-out evaluation (10,000) subsets. An MD5-based deduplication mechanism ensures that no instruction pair appears in both subsets. This prevents exact-pair leakage but does not preclude structural similarity across different instantiations of the same pattern—a known limitation of procedurally generated benchmarks that is discussed in Section 9.3. CODE-2 exercises have determinate solutions (register assignments are prescribed by the problem statement), making a fixed evaluation protocol valid for this domain.

5 Model Optimization and Adaptation Strategies

5.1 Model Selection: T5 as a Translation Architecture

We selected the T5-Small encoder-decoder architecture (60 million parameters) [6]. While decoder-only models are popular for open-ended code completion [12], T5's encoder-decoder structure is better suited to our task: we are not completing text but *translating* a functional description into a formal grammar. The encoder constructs a dense representation of the student's natural-language query; the decoder synthesizes the assembly sequence token-by-token [13]. T5-Small's 60M parameter count is also appropriate for CPU-only inference: it fits comfortably within the memory constraints of laboratory workstations after INT8 quantization.

5.2 Comparative Evaluation of Fine-Tuning Strategies

A significant engineering decision was the choice of adaptation strategy. We evaluated three methods on a consumer GPU (NVIDIA RTX 2060, 6 GB VRAM):

5.2.1 Prompt Tuning (Lightweight Baseline)

Following Lester et al. [14], we froze all 60M model parameters and appended 100 learnable soft prompt tokens to the input embedding. This method updates fewer than 0.1% of parameters and requires minimal GPU memory. However, soft prompts were insufficient to override the model's pre-trained knowledge of standard x86 and ARM assembly syntax. The model consistently hallucinated instructions from irrelevant ISAs, failing to adhere to CODE-2 constraints. Prompt Tuning is effective when the base model already has domain overlap; CODE-2 is absent from any pre-training corpus, making this approach inadequate.

5.2.2 Low-Rank Adaptation (LoRA)

Following Hu et al. [15], we injected trainable low-rank matrices ($r = 8$, $\alpha = 32$) into the attention layers while keeping the backbone frozen. LoRA substantially outperformed Prompt Tuning (Table 1) and produced syntactically coherent sequences in most cases. However, qualitative analysis of inference outputs revealed systematic failures on complex branching instructions requiring precise operand ordering and label scoping. The low-rank bottleneck appears to limit capacity for learning a rigid formal grammar from scratch when no relevant prior knowledge exists in the base model.

5.2.3 Full Fine-Tuning

We unfroze all 60M parameters and trained the complete network on the CODE-2 task bank. Hyperparameters were optimized using the Optuna framework [16]: a search over learning rates (10^{-5}–10^{-3}) and batch sizes (8, 16, 32) identified the optimal configuration (learning rate: 2.8×10^{-4}, batch size: 16, AdamW optimizer). Full Fine-Tuning required more GPU memory and training time than LoRA but achieved substantially higher Exact Match, as reported in Section 6. This outcome is consistent with the hypothesis that adapting a model to a fully novel formal language requires complete weight updates.

6 Technical Evaluation

6.1 Metrics and Rationale

The operational requirement for an assembly scaffolding tool is strict: code that does not assemble correctly provides no benefit to the student. Accordingly, we evaluate primarily with

Exact Match (EM), i.e., the share of outputs that match the reference solution character-for-character. Because CODE-2 exercises specify exact registers and memory locations, equivalent-but-different solutions are not expected in the benchmark; EM therefore best reflects deployability. BLEU and ROUGE-L are reported as secondary metrics to provide a diagnostic view of partial similarity.

6.2 Fine-Tuning Strategy Comparison

Table 1 reports performance on the held-out evaluation set ($n = 10{,}000$, representing the 20% split).

Table 1: Adaptation strategy performance on the held-out evaluation set ($n = 10{,}000$ samples, 20% of total dataset). EM = Exact Match. Higher is better for all metrics.

Adaptation Method	**EM (%)**	**BLEU**	**ROUGE-L**
Prompt Tuning	45.20	0.821	0.845
LoRA ($r = 8$, $\alpha = 32$)	69.25	0.971	0.980
Full Fine-Tuning	**94.10**	**0.995**	**0.998**

The 94.10% EM achieved by Full Fine-Tuning confirms that this method satisfies the deployability threshold for a supervised scaffolding tool. The contrast between LoRA's BLEU score (0.971) and its EM (69.25%) illustrates a critical engineering lesson: near-surface similarity does not imply syntactic correctness for assembly code, where a single transposed operand constitutes a complete functional failure.

The finding that LoRA underperforms Full Fine-Tuning for rigid assembly ISA adaptation provides actionable guidance for practitioners: when the target formal language is absent from the base model's pre-training distribution, lightweight PEFT methods may be insufficient, and the additional computational cost of Full Fine-Tuning during the (offline) training phase is justified.

6.3 Quantization for CPU-Only Deployment

To deploy the model on standard laboratory hardware, we applied post-training INT8 quantization [17] via ONNX Runtime. This maps 32-bit floating-point weights to 8-bit integers, reducing model size and improving inference throughput. Table 2 summarizes the resulting latency, model size, and Exact Match performance.

Table 2: Inference performance on a representative laboratory workstation (Intel i5, 8 GB RAM, no GPU). Latency is the mean over 500 inference calls on the evaluation set. EM is reported on the same held-out set as Table 1.

Model Version	Latency (ms)	Size (MB)	EM (%)
Full FT, FP32 (PyTorch)	1,689	242	94.10
Full FT, INT8 (ONNX)	**526**	**68**	**93.25**

The INT8 model reduces inference latency by 69% (1,689 ms $\rightarrow$ 526 ms) and model size by 72% (242 MB $\rightarrow$ 68 MB), with an EM reduction of less than 1 percentage point (94.10% $\rightarrow$ 93.25%). Bringing response time below 600 ms is relevant to laboratory usability: it eliminates the perceptible delay that would otherwise interrupt student workflow. The negligible precision loss confirms that INT8 quantization represents a favorable engineering trade-off for this deployment context.

6.4 Robustness Stress Test on the Deployable INT8 Pipeline

To address robustness under realistic student input noise (typographical errors, truncated intents, and Spanish–English code-mixing), we conducted a stress test on the deployable INT8 pipeline over $n = 2{,}000$ samples; results are summarized in Table 3. The main evaluation (EM=0.941) was obtained on the full test set used in the manuscript's primary benchmark protocol. The robustness stress test (clean EM=0.830) reuses the same preprocessing and decoding pipeline on a repository subset designed for input-perturbation analysis; the remaining gap is consistent with distributional differences between the stress test subset and the primary benchmark set.

Table 3 reports results. Performance remains high on clean inputs (EM=0.830) and is competitive under code-mixing (EM=0.739), a perturbation type that closely mirrors the bilingual context of the student population. Typographical noise (EM=0.0285) and truncated prompts (EM=0.0030) constitute clear failure modes; these are expected boundaries for a model trained exclusively on well-formed synthetic queries. Crucially, the OOS opcode rate is 0.000 across all conditions, indicating that the model does not hallucinate instructions outside the CODE-2 instruction set under any evaluated perturbation.

Quantization impact was additionally measured on a matched subset ($n = 300$): EM decreases from 0.8033 (FP32) to 0.7967 (INT8), $\Delta\text{EM} = -0.0067$. Although FP32 and INT8 outputs differ frequently (disagreement rate=0.448), out-of-spec generation remains near-zero (`disagreement_with_OOS`=0.001), confirming that INT8 affects low-margin token selection rather than introducing systematic errors.

Table 3: Robustness stress test on the deployable INT8 pipeline ($n = 2{,}000$). Exact Match (EM) under input perturbations and out-of-spec opcode rate (OOS) computed relative to the mnemonic set present in the evaluated split `{LD, ST, ADDS, SUBS}`.

Condition	INT8 EM	OOS opcode rate
Clean	0.830	0.000
Typographical noise	0.0285	0.000
Ambiguous (truncated)	0.0030	0.000
Code-mix (ES/EN)	0.739	0.000

7 Laboratory Deployment and Integration

7.1 Deployment Context

The system has been integrated into a Computer Architecture laboratory course serving approximately 120 students per semester. Laboratory sessions are conducted on standard desktop workstations; internet access is restricted during examinations and certain assessed practicals, making local deployment a functional requirement. The assistant operates as an optional resource available during open laboratory hours, not as a component of formal assessment.

7.2 Supervised Integration Workflow

The assistant is explicitly positioned as a syntax-level scaffolding aid, not a solution oracle. The integration follows a three-stage supervised protocol:

1. **Independent attempt:** Students must first attempt the exercise using course materials and their own knowledge.
2. **Scaffolded query:** Students blocked by a specific syntactic uncertainty (e.g., the operand order for `SUBS`) may query the assistant. The assistant generates a syntactic scaffold.
3. **Mandatory emulator verification:** The student copies the scaffold into the CODE-2 emulator, executes it step-by-step, and documents the resulting register and flag states. No credit is awarded without verified execution.

This workflow positions the AI as a "junior drafter" subject to verification by the student as the accountable engineer—a framing that reflects professional practice with AI-assisted development [7].

8 Pedagogical Implications

The following implications are presented as plausible hypotheses grounded in pedagogical theory. No controlled learning-gains study has been conducted, and no claims of measured educational

impact are made. These observations are intended to motivate future empirical work.

8.1 Reducing the Blank Page Barrier

By providing a structurally correct starting point, the assistant may lower the cognitive barrier to initiating an assembly task. Rather than allocating working memory to mnemonic recall, students can concentrate on verifying and understanding the generated sequence [2]. This is consistent with scaffolding theory [4]: temporary support is provided at the point of need and withdrawn as competence develops (emulator verification ensures active engagement rather than passive copying).

Operationally, mandatory emulator-based verification implements scaffolding fading: the assistant provides an initial syntactic draft, while responsibility progressively shifts to the learner through step-by-step state inspection and correction.

8.2 The AI Auditing Protocol as a High-Order Activity

The system's non-zero error rate (≈6% on the held-out set) is exploited as a pedagogical resource. We have introduced "AI Auditing" activities in which students receive a problem and an AI-generated scaffold containing a deliberate semantic error:

```
; Student intent: R2 = R2 - R1
  SUBS R1, R1, R2 ; Error: incorrect destination register
```

Listing 2: Type-B error used in AI Auditing activities: destination and source operands are swapped.

The student's task is to identify and correct the error. This activity shifts the cognitive goal from *Application* to *Evaluation* in Bloom's revised taxonomy, fostering critical engagement with automated tools [7]—a competency increasingly relevant as AI-assisted programming tools become standard in the industry.

9 Discussion

9.1 Infrastructure Scalability

Preliminary classroom observation (informal, not a controlled study) suggests a reduction in routine syntax-level queries directed to instructors during laboratory sessions. If borne out by formal measurement, this would represent a meaningful improvement in instructor allocation: teaching staff could focus on higher-order conceptual difficulties (interrupt handling, data path design) while the assistant handles syntax reminders. This scalability argument is the primary motivation for the infrastructure investment, not learning outcomes per se.

9.2 Transferability of the Pipeline

The engineered pipeline (procedural data generation → Full Fine-Tuning → INT8 quantization → CPU deployment) is domain-agnostic. Engineering disciplines relying on other rigid formal languages—VHDL for FPGA design, IEC 61131-3 ladder logic for industrial PLCs, G-code for numerical control—could apply this methodology to create specialized, privacy-preserving laboratory assistants. The key design principle is that the target language must be fully specifiable by a finite set of pedagogical patterns, enabling synthetic data generation in the absence of a real-world corpus.

9.3 Limitations

Several limitations constrain the conclusions of this work:

1. **Synthetic data distribution:** The evaluation set is drawn from the same procedural generator as the training set. While MD5-based deduplication prevents exact-pair leakage, structurally similar pairs (different register instantiations of the same pattern) may appear in both subsets. Generalization to student queries not covered by the 200 patterns has not been formally evaluated.

2. **No learning-gains data:** We report no pre/post tests, no control group, and no longitudinal assessment. Pedagogical benefits are hypothesized, not demonstrated. A controlled study is planned as future work.

3. **Evaluation set terminology:** The 20% split used for evaluation was not held out from hyperparameter selection in all cases; it functions more accurately as a held out evaluation set than a true test set. Future work should employ a three-way split (train/validation/test).

4. **Single institution:** Deployment is at one institution with one student cohort. Generalization to other architectures, curricula, and student populations requires further study.

5. **Input robustness:** As quantified in Section 6.4 (Table 3), typographical noise (EM=0.0285) and truncated prompts (EM=0.0030) represent operational constraints under the current training distribution; robustness to out-of-distribution student input is planned as future work.

9.4 AI Literacy and Engineering Responsibility

By framing the assistant as a supervised tool requiring emulator-based verification, the integration workflow teaches engineering responsibility: probabilistic models are fallible, and final accountability for correctness lies with the human engineer. This mirrors professional practice with tools such as GitHub Copilot [7] and is an essential competency for engineers entering AI-augmented workplaces.

10 Conclusion

This paper presents the design, implementation, and technical evaluation of a locally deployable GenAI assistant for assembly language scaffolding in the CODE-2 educational environment. The primary contribution is a replicable infrastructure pipeline—synthetic data generation, Full Fine-Tuning, INT8 quantization, and CPU deployment—that enables scalable syntax-level support in CPU-only laboratory settings without internet dependency or cloud data exposure.

Technical results confirm that Full Fine-Tuning achieves 94.10% Exact Match on the held-out evaluation set, that lightweight PEFT methods (LoRA, Prompt Tuning) are insufficient for formal-language adaptation when the target ISA is absent from the base model's pre-training distribution, and that post-training INT8 quantization reduces inference latency by 69% with negligible precision loss, bringing response times within acceptable limits for laboratory use.

Pedagogical benefits—including reduced cognitive load during syntax tasks and the development of critical AI evaluation skills—are presented as plausible implications of the infrastructure design. A controlled longitudinal study to quantify learning gains is planned as immediate future work. The extension of the synthetic task bank to cover advanced interrupt-driven I/O and the pipeline's transferability to other rigid formal languages (VHDL, PLC ladder logic) are additional directions for future investigation.

Data and Code Availability

The Synthetic Task Bank generator scripts, the fine-tuned T5-Small models in ONNX INT8 format, and the local laboratory server code will be made publicly available in a persistent repository upon acceptance of this manuscript to support reproducibility in the engineering education community. An anonymized supplementary archive has been provided with this submission for reviewer access.

Author Contributions

[Blinded for double-blind peer review.]

Author 1: Conceptualization, methodology, software and infrastructure, data curation, formal analysis, validation, visualization, writing—original draft, writing—review and editing.

Acknowledgements

[Blinded for double-blind peer review. Full acknowledgments, including funding sources, institutional affiliations, and course participant recognition, will be restored in the accepted manuscript.]

Conflict of Interest

The authors declare no conflicts of interest relevant to this work.

References

[1] M. Ben-Ari, "Constructivism in computer science education," *ACM SIGCSE Bulletin*, vol. 30, no. 1, pp. 257–261, 1998.

[2] J. Sweller, "Cognitive load during problem solving: Effects on learning," *Cognitive Science*, vol. 12, no. 2, pp. 257–285, 1988.

[3] K. Kirshenbaum, E. S. Wiese, and N. Thota, "Dynamic scaffolding in computer architecture," in *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE)*, pp. 44–50, 2020.

[4] O. Zawacki-Richter, V. I. Marín, M. Bond, and F. Gouverneur, "Systematic review of research on artificial intelligence applications in higher education—challenges and opportunities," *International Journal of Educational Technology in Higher Education*, vol. 18, no. 1, pp. 1–42, 2021.

[5] Z. Liu, J. Tang, Y. Zhu, *et al.*, "A survey on large language models for code generation," *ACM Computing Surveys*, 2024.

[6] C. Raffel, N. Shazeer, A. Roberts, *et al.*, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020.

[7] J. Prather *et al.*, ""it's weird that it knows what i want": Usability and interactions with copilot for novice programmers," *ACM Transactions on Computer-Human Interaction*, vol. 31, no. 1, pp. 1–31, 2023.

[8] A. Prieto, F. J. Pelayo, F. Gómez-Mula, J. Ortega, A. Cañas, A. Martínez, and F. J. Fernández, "Un computador didáctico elemental (CODE-2)," in *Actas de las VIII Jornadas de Enseñanza Universitaria de la Informática (JENUI 2002)*, (Cáceres, Spain), pp. 117–124, Universidad de Extremadura, 2002.

[9] D. Sculley, G. Holt, D. Golovin, *et al.*, "Hidden technical debt in machine learning systems," in *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 2503–2511, 2015.

[10] T. Kudo and J. Richardson, "SentencePiece: A simple and language-independent subword tokenizer and detokenizer for neural text processing," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations (EMNLP)*, pp. 66–71, 2018.

[11] ONNX Community, "ONNX: Open neural network exchange." `https://onnx.ai`, 2023. Accessed: 2025-09-01.

[12] M. Chen, J. Tworek, H. Jun, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[13] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[14] B. Lester, R. Al-Rfou, and N. Constant, "The power of scale for parameter-efficient prompt tuning," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 3045–3059, 2021.

[15] E. J. Hu, Y. Shen, P. Wallis, *et al.*, "LoRA: Low-rank adaptation of large language models," in *International Conference on Learning Representations (ICLR)*, 2022.

[16] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 2623–2631, 2019.

[17] M. Nagel, M. Fournarakis, R. A. Amjad, *et al.*, "A white paper on neural network quantization," *arXiv preprint arXiv:2106.08295*, 2021.