Workflows vs Agents for Code Translation

Anonymous Author(s)
Affiliation
Address
email

Abstract

Translating algorithms from high-level languages like MATLAB to hardware description languages (HDLs) is a resource-intensive but necessary step for deployment on FPGAs and ASICs. While large language models (LLMs) offer a path to automation, their limited training on HDL code makes end-to-end transpilation brittle and prone to syntax errors. We compare two LLM-driven methods for syntax repair in a MATLAB-to-HDL pipeline: a structured, expert-designed flow that follows a fixed sequence of operations, and a more autonomous agentic approach that uses the Model Context Protocol (MCP) [1] to dynamically select its own tools. We study 42 MATLAB signal-processing functions and isolate the syntax-repair stage. Across three model scales, the agentic approach is more effective at resolving initial syntax errors, unblocking a greater number of candidates to proceed through the pipeline. This upstream improvement yields measurable downstream improvements, most notably on mid-sized models, where it increases the simulation reach rate by over 20 percentage points. We hypothesize the gains come from short prompts, aggressive context management, and conditional tool use. Conditional retrieval helps at 8B and 30B; at 235B final-success gains are small and a naive RAG variant attains the highest final success. Our findings suggest that these agentic frameworks, when properly designed, are most effective at compensating for the capacity limits of small and mid-sized models.

1 Introduction

Most digital signal processing algorithms are at least initially developed in MATLAB because it offers rapid iteration, a rich set of operations and toolboxes, and convenient testbench generation. However, deployment targets are often FPGAs or ASICs that demand code written in a hardware description language for low latency, high throughput, tight power budgets, and greater resource control. Manually bridging the gap between these paradigms, from MATLAB's high-level, dynamically typed environment to the low-level, statically typed structure of an HDL, is known to be slow and error-prone. Traditional compiler approaches, which are rule-based, are limited in their application to a subset of operations and rigid in the code format they generate.

Although large language models (LLMs) promise to accelerate this process, their proficiency with HDLs is limited by the scarcity of high-quality open-source training data. As a result, direct end-to-end translation often fails, requiring a structured pipeline with robust guardrails. In such pipelines, failures frequently occur in the initial syntax repair stage. Naive, automated fixes at this step can silently alter the code's semantics, ultimately leading to verification failures.

In this paper, we focus on this critical syntax repair stage and conduct an empirical comparison of two LLM-driven strategies. We evaluated an expert-designed flow that follows a fixed script against a flexible agentic framework that uses the Model Context Protocol (MCP) to dynamically select tools. We study 42 MATLAB signal-processing functions and isolate the syntax-repair stage.

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

This paper makes three primary contributions. First, we provide a detailed empirical comparison of expert-designed and agentic flows across three model scales, quantifying their impact on pipeline success. Second, from these results, we derive a set of practical design guidelines for building effective agentic frameworks, emphasizing the importance of minimal prompts and aggressive context management. Finally, we isolate and demonstrate a key retrieval strategy, showing that on-demand tool use helps at small to mid scale, but naive inclusion of the same tools is actively detrimental to performance.

Beyond the MATLAB to HDL setting, these results expose a broader pattern in agentic code systems. The experiments suggest that performance depends less on the presence of tools than on how and when they are invoked. Conditional retrieval, prompt minimalism, and separation of planning from generation emerge as scale-sensitive design levers that generalize to other agentic workflows such as refactoring, type inference, and translation between low-resource languages. We therefore view syntax repair as a controlled case study revealing general principles for reliable tool use in language-model-driven programming.

2 Related Work

Our work is situated at the intersection of several key research areas: LLMs for code translation, agentic frameworks with tool use, and the specific challenge of translating code to hardware description languages (HDLs).

The use of large language models for programming tasks was measurably advanced by models like Codex, which demonstrated a strong ability to generate code in common languages [2]. However, these models often struggle with the "long tail" of specialized or low-resource languages. Recent work by Vijayaraghavan et al. (2024) specifically highlights this, showing that while LLMs can generate functionally correct VHDL, they often produce non-synthesizable or inefficient code that requires major manual correction [11]. This motivates our focus on a structured pipeline with a dedicated repair stage, rather than relying on direct, end-to-end translation.

Several recent efforts have focused specifically on the MATLAB-to-HDL problem. For instance, Schwartz et al. (2024) introduced a fine-tuning approach to improve an LLM's proficiency in VHDL, demonstrating gains but also noting the high cost of data acquisition [9]. Concurrently, Thakur et al. (2023) developed VeriGen, a system that uses an LLM to translate Python to Verilog for specific dataflow applications, relying on formal methods to constrain the output [10]. Our work complements these approaches by focusing not on model fine-tuning or formal constraints, but on a more flexible, agentic repair process that can be applied to general-purpose LLM outputs.

To overcome the limitations of standalone LLMs, recent research has focused on agentic frameworks that allow models to use external tools. Foundational work like ReAct [12] established the core "reason-act" loop, while Toolformer [8] showed how models could learn to use APIs. Our MCP-based flow builds directly on these concepts, providing the LLM with a set of specialized tools (a compiler, a retrieval system) to diagnose and fix errors. A key contribution of our paper is the analysis of the *strategy* for tool use. While the use of Retrieval-Augmented Generation (RAG) [4] is a common technique, we demonstrate that for code repair, a conditional invocation policy is critical to its success.

3 Experimental Setup

To evaluate the effectiveness of an agentic framework in a real-world programming task, we integrated two distinct syntax repair methodologies into a MATLAB-to-HDL transpilation pipeline[1]. This setup allowed us to compare not only their immediate success in repairing syntax but also their impact on downstream verification and final code quality.

3.1 Transpilation Pipeline

Our end-to-end pipeline, illustrated in Figure 1, begins by generating a testbench from the source MATLAB code that is used to verify the behavior of the HDL code generated. If necessary, the MATLAB code is then converted to fixed-point arithmetic and to handle data in a streaming rather

[1]Transpilation commonly stands for translating compilation, a form of language to language translation.

than batch fashion. An LLM uses this version to generate multiple candidate HDL translations (in our case, 3). Because this initial translation frequently introduces errors, each candidate is sent to a dedicated syntax repair stage. This is the stage that we isolate for our experiment. After repair, syntactically correct code proceeds to a synthesis stage and is finally validated against the original testbench.

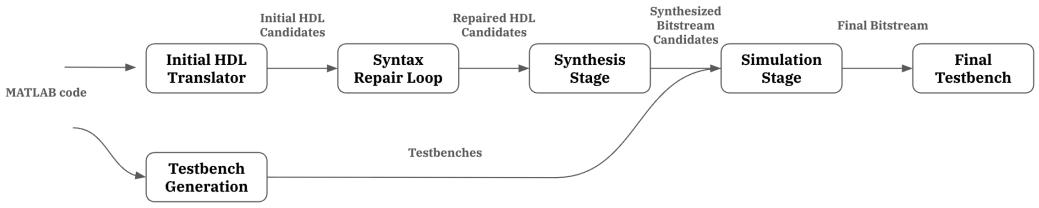

Figure 1: MATLAB to HDL, end-to-end pipeline

3.2 Comparing Syntax Repair Flows

Expert-Designed Flow (Baseline): As shown in Figure 2, this flow represents a structured, non-agentic approach. For each HDL candidate, the LLM is provided with a large, expert-written prompt containing detailed guidance and advice for repairing HDL syntax. It uses the GHDL compiler to identify errors and iterates on the code in a fixed loop until syntax passes or a limit is reached.

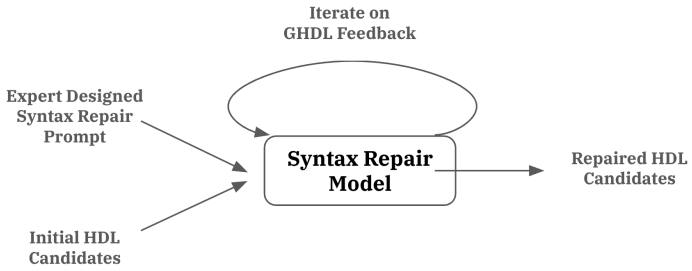

Figure 2: The expert-designed baseline flow, which follows a fixed repair script.

Agentic MCP Flow: In contrast, the flow in Figure 3 provides the LLM with autonomy. The agent receives a minimal prompt containing only the broken code, a repair goal, and a menu of available tools. It is then free to independently select and sequence tools to solve the problem. The tools provided were the following.

1. **GHDL Syntax Check:** The same compiler used in the baseline flow to get a list of syntax errors.
2. **RAG Retrieval:** A tool to retrieve syntactically correct VHDL code examples from a vector database to provide relevant context.
3. **Code Rewrite:** A tool that passes the original code and a set of model-generated instructions to a second clean context agent for implementation. This design avoids context contamination from previous failed attempts.

The context was aggressively pruned to maintain performance and avoid exceeding the context window of smaller models. This decision was motivated by preliminary experiments showing that including history from failed repair iterations decreased performance. Therefore, the context was reset after each attempt, with only a brief summary of the previous attempt carried over.

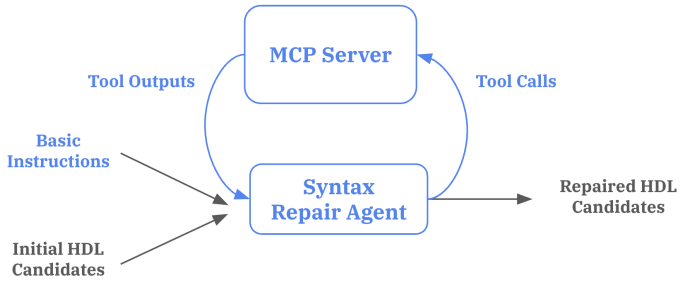

Figure 3: The agentic MCP flow, where the agent dynamically selects tools in an iterative loop.

3.3 Dataset and Evaluation Metrics

We evaluated both flows on an internal dataset of 42 MATLAB functions for signal processing, ranging from 3 to 311 lines of code. We measured performance at both the candidate and function levels across three key metrics:

1. **Syntax Pass Rate:** The percentage of candidates (or functions with at least one candidate) that successfully pass the GHDL syntax check.
2. **Simulation Reach Rate:** The percentage of functions whose repaired code is valid enough to proceed to the synthesis and simulation stage.
3. **Final Flow Success Rate:** The percentage of functions whose final HDL output passes the MATLAB-generated testbench, confirming semantic equivalence.

3.4 RAG Corpus and Retrieval Configuration

We built a retrieval corpus of syntactically correct VHDL functions drawn from internal examples used in class exercises and small design blocks. More than 1000 functions were included for reference by the tool. The functions were short, generally under 100 lines. Each item was stored as a single document representing a complete function, not fragmented into overlapping chunks. This whole function policy preserved idioms such as library imports, type declarations, and process structure that matter for synthesis.

We embedded each document using Infinity embeddings [3]. At query time the agent formed a text query from the current error context and the working VHDL variant, then retrieved the top 3 nearest neighbors. The examples were inserted verbatim as full functions, subject to a strict token budget with truncation rules that favored keeping headers, library lines, and process scaffolding.

In the naive Non MCP with RAG variant, the top 3 examples were appended unconditionally to every repair prompt. In the MCP variant, retrieval was conditional on lack of progress or on compiler errors that indicated missing idioms, and the agent could choose to skip retrieval. No additional reranking by compiler error similarity was used.

We used three Qwen3 checkpoints from Hugging Face, Qwen3-8B [7], Qwen3-30B-A3B [6], and Qwen3-235B-A22B [5], with default tokenizers and no additional fine-tuning.

3.5 Reproducibility.

Additional configuration and decoding details are provided in Appendix B.

4 Experimental Results

We evaluated the expert-designed (non-MCP) and agentic (MCP) syntax repair flows on our 42-function MATLAB dataset using Qwen across three model scales: 8B, 30B, and 235B. The results

demonstrate that the agentic MCP approach consistently improves pipeline progression, with the most measurable impact observed at the mid-scale 30B model.

Variability across twelve independent runs was small ($\sigma \approx$ 2-3 pp on intermediate metrics in pilot logging).

4.1 MCP Measurably Improves Pipeline Progression at 30B

At the 30B scale (Table 1), the agentic MCP approach yields substantial improvements over the expert-designed baseline across all intermediate metrics. The candidate-level syntax pass rate increases from 51.9% to 75.0% (+23.1 pp), and the function-level syntax pass rate increases from 81.2% to 92.3% (+11.1 pp). The most dramatic gain is in the share of functions that reach the simulation stage, which jumps from 72.1% to 95.3% (+23.2 pp). This upstream success translates into a measurable improvement in the end-to-end success rate, which improves from 33.5% to 42.1% (+8.6 pp). These results indicate that while downstream semantic issues remain a bottleneck, MCP is highly effective at resolving the initial syntax errors that cause the most pipeline attrition. For reference, the naive variant Non-MCP + RAG underperforms at 30B; see Table 1, where the simulation reach is 44.0% and the final success is 19.5%.

4.2 The Effect of MCP is Scale-Dependent

The benefits of the agentic framework vary with the size of the model, as shown in Tables 2 and 3.

- **At 8B**, MCP provides a crucial lift for the smaller model, improving the function-level syntax pass from 76.7% to 90.7% (+14 pp) and boosting the simulation reach rate by over 20 pp. However, the model's limited capacity constrains its ability to translate these gains into end-to-end success, which sees a more modest improvement (+4.9 pp).

- **At 235B**, the baseline expert-designed flow is already highly competent, successfully repairing syntax for 93% of functions. Here, MCP has less headroom to add value. It pushes the function-level syntax pass to 100% and provides a small lift to the final success rate (+2.3 pp), but its overall impact is diminished. In particular, the naive variant non-MCP + RAG attains the highest final success at 235B (58.1% vs 55.8% for MCP; Table 3); we defer the analysis to Section 5.

4.3 Conditional Tool Use is Critical; Naive RAG Inclusion Degrades Performance

To isolate the impact of tool-use policy, we tested a variant that naively appended RAG outputs to every repair prompt. As shown in the Non-MCP+RAG columns of Tables 2, 1, and 3, unconditional inclusion is detrimental at smaller and mid scales. At the 30B scale (Table 1), naively adding RAG caused the simulation reach rate to drop from 72.1% to 44.0% and the final success rate to drop from 33.5% to 19.5%. This provides strong evidence that the agentic framework's success is driven not just by the availability of tools, but by its ability to apply them conditionally and avoid the context clutter that harms less capable models.

Table 1: Qwen 30B, function-level macro averages (baseline, MCP, and baseline + naive RAG).

Metric	Non-MCP	MCP	Non-MCP+RAG
Candidate-level syntax pass	51.9%	**75.0%**	60.0%
Function-level syntax pass	81.2%	**92.3%**	77.0%
Reach testbench	72.1%	**95.3%**	44.0%
Final success	33.53%	**42.12%**	19.5%

Table 2: Qwen 8B, function-level macro averages (baseline, MCP, and baseline + naive RAG).

Metric	Non-MCP	MCP	Non-MCP+RAG
Candidate-level syntax pass	59.0%	**63.1%**	56.7%
Function-level syntax pass	76.7%	**90.7%**	76.7%
Reach testbench	60.5%	**90.7%**	60.5%
Final success	18.3%	**23.2%**	16.9%

Table 3: Qwen 235B, function-level macro averages (baseline, MCP, and baseline + naive RAG).

Metric	Non-MCP	MCP	Non-MCP+RAG
Candidate-level syntax pass	86.0%	**94.4%**	93.9%
Function-level syntax pass	93.0%	**100%**	100%
Reach testbench	100%	100%	100%
Final success	53.5%	55.8%	**58.1%**

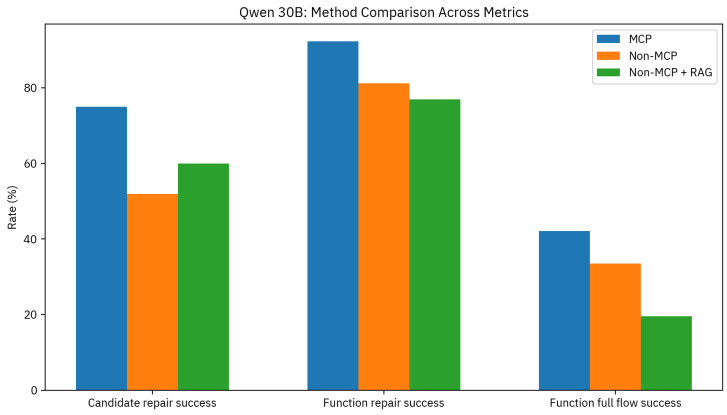

Figure 4: metrics comparison for 30B model

5 Discussion

5.1 Conditional Tool Use is the Primary Driver of Improvement

Our results show that an agentic framework using MCP for syntax repair measurably improves pipeline progression compared to a deterministic, expert-designed flow. The success of this approach, however, depends heavily on a set of core design principles and the underlying model's scale.

To understand why MCP helps, we consider how the framework structures reasoning within the context window. We hypothesize that these improvements arise from differences in prompt entropy and token allocation rather than from the tools themselves. When the model must process long, unfiltered retrieval outputs, its effective reasoning bandwidth is diluted. Conditional invocation instead maintains a compact, high-signal context that fits within the model's limited working memory. While we did not directly test this mechanism, the scale pattern in Tables 1-3 is consistent with this interpretation.

The central finding of our work is that how auxiliary information is introduced matters as much as what information is available. At the 30B scale, the agentic approach delivered large improvements to intermediate metrics, most notably lifting the simulation reach rate from 72.1% to 95.3%. This gain is directly attributable to a conditional tool-use policy.

This policy was encoded in a prompt that created a simple "if stuck, then retrieve" loop, instructing the agent to first attempt a local fix and only invoke RAG when it failed to make progress. We found

that RAG helps when an error requires nonlocal structure or idioms absent from the current context, but it hurts when it injects loosely related code that widens the search space or distracts the model.

This was validated by our negative control experiment, where naively appending RAG outputs to the expert-designed flow degraded performance sharply. At 30B, this unconditional retrieval caused the final success rate to drop from 33.5% to 19.5%. We identified three failure modes for this approach: context clutter from large examples, architectural mismatches in retrieved code, and the truncation of precise compiler errors. The agentic framework's ability to selectively deploy tools avoids these pitfalls and is therefore its key advantage.

This pattern suggests that agentic frameworks contribute value not by adding new capabilities, but by enforcing structure that protects limited attention within the model's context window. This observation aligns with prior findings on reasoning bandwidth limits in LLMs and suggests that agentic orchestration may act as a form of context regularization.

5.2 The Interplay of Agentic Design and Model Scale

The effectiveness of the MCP framework is clearly dependent on the model's intrinsic capacity. We observed two forces interacting with scale: the planning and selectivity which MCP supplies, and the intrinsic capacity which depends on the model's size.

When capacity is low (8B), MCP's structure enables progress but cannot fully compensate for limited semantic modeling. It provides a crucial boost to pipeline progression but only a modest lift to final success.

When capacity is moderate (30B), MCP's selective tool use and context hygiene are complementary, yielding the greatest overall uplift. The model has enough reasoning bandwidth to exploit the framework's design, converting a large number of otherwise failing candidates into successful simulations.

When capacity is high (235B), the model's intrinsic ability saturates progression metrics, leaving only a narrow semantic frontier where MCP can help. The framework's benefit is real but small relative to its overhead.

This pattern suggests that agentic frameworks are most impactful at the mid-scale, where they effectively compensate for a model's capacity limits without being rendered redundant by its sheer competence.

These findings indicate that policy of tool use is a controllable design variable. Conditional, selective invocation yields greater benefit at smaller model scales, whereas naive inclusion suffices or even helps at extreme scale. The practical takeaway is that reliable agentic systems depend as much on disciplined orchestration as on model capacity. Although the present study isolates the syntax-repair stage, the same orchestration principles-conditional tool use and separation of planning from generation likely extend to broader software-engineering workflows where LLMs operate under token or attention constraints.

5.3 Naive RAG at 235B

At 235B, the naive Non MCP with RAG variant attains the highest end to end success (58.1% versus 55.8% for MCP; Table 3). Intermediate metrics are saturated at this scale. Function level syntax pass is 100% for both variants, and reach to testbench is 100% for all methods. The gain therefore most likely arises at the final verification stage through canonicalization rather than earlier unblocking.

A simple hypothesis explains the pattern. Larger models filter non helpful retrieved tokens and are less susceptible to context saturation. Naive retrieval then acts as few shot priming instead of a distraction. In smaller and mid scale models, MCP helps by keeping prompts short and focused. At 235B that focusing benefit is largely endogenous to the model.

The practical implication is a hybrid policy for large models. Attach a compact deduplicated top k exemplar set that persists across attempts, and keep MCP for compiler guided diagnosis. Enforce strict length control to preserve signal to noise. The observed advantage is small at 2.3 percentage points, so it may fall within run to run variance.

We treat this as a testable claim. Run length controlled ablations with equal token budgets, inject noise into retrieval to probe filtering, rerank retrieval by compiler error similarity, disable MCP resets to test persistence effects, and report confidence intervals to establish whether the gain is statistically reliable.

These numbers (Tables 2, 1, 3) summarize the scale pattern: naive Non–MCP+RAG harms 8B and 30B but attains the highest final success at 235B; we treat the filtering and context–saturation explanation as a hypothesis pending length–controlled ablations and CIs.

5.4 Design evolution and lessons

Our design converged through four stages. The first prototype reused the prior framework that received the full VHDL and was instructed to output new VHDL in `<vhdl>` tags. We added a small tool set: a manual lookup, a similar examples retrieval tool, and a helper that proposed subproblems. The agent called tools directly, accumulated all tool outputs in the prompt, and deferred code generation. This version rarely produced code before the context filled. Prompt tuning reduced tool calls, but performance still lagged the expert script.

The second stage allowed the agent to interleave thinking and tool calls and to place tool outputs into a persistent reasoning trace, then generate. Syntax pass improved, but context length again became the bottleneck.

The third stage switched to a suggest change policy. The agent proposed edits at specific line numbers, with the latest VHDL always shown in context. We added visible line numbers and required each edit to name the lines and include a tight local window around the change. This reduced tokens and worked on small refactors, but it failed when many declarations or process blocks needed coordinated changes.

The final stage separated analysis and generation. A tool using planner produced a compact instruction list and short rationale in a clean format. A second generator, with a clean context, produced a complete new VHDL unit from those instructions. This separation increased candidate level syntax pass and function level reach across scales. It also made behavior more stable, since the generator never saw long tool transcripts or prior failed attempts.

Tool ablations informed the final menu. The manual lookup tool reduced performance whenever it was invoked, so we removed it. The similar examples retrieval tool helped within MCP when used conditionally and under a strict token budget, but naive attachment to every prompt reduced performance at 8B and 30B while slightly helping at 235B. This pattern supports a policy conclusion rather than a tool conclusion: conditional use matters more than tool availability.

Why this worked: separating planning from generation reduced context contamination and kept token budgets predictable; short instruction payloads preserved signal to noise; and resets avoided anchoring on earlier failed edits. These mechanics align with the observed scale effects. At small and mid scale, MCP limits distraction and improves reach. At large scale, model capacity filters non helpful tokens so naive retrieval can act as few shot priming, which explains the small end to end gain at 235B.

For reproduction, report the corpus construction in Section 3.4, the embedding model identifier, the top k used for retrieval, any deduplication, and fixed token budgets for planner instructions and inserted examples. Equalize token budgets when comparing policies to separate policy effects from length effects.

5.5 Limitations and Future Work

Our dataset consists of 42 functions from a single domain (signal processing), so these results may differ with other error profiles. The primary improvements we observed were in preventing pipeline attrition before the final verification stage, which indicates that a measurable semantic bottleneck remains downstream of syntax repair. Future work should focus on pairing agentic repair with semantic safeguards, such as differential testing or lightweight equivalence checks, to ensure that the gains from improved syntax repair fully propagate to the end-to-end success rate.

The dataset used in this study is currently internal to our company. We plan to release a cleaned subset of the 42 MATLAB functions and corresponding HDL outputs in the near future to support external replication and follow-up work.

Broader Impacts

This work studies agentic tool-use for program repair and translation. Although evaluated on MATLAB→HDL, the design lessons (conditional retrieval, context control, separation of planning and generation) apply to agentic programming workflows such as refactoring, porting, linting, and large-scale code health tasks. Potential benefits include faster prototyping, lower entry barriers for hardware and systems development, and safer automation compared to unconstrained prompting.

Risks include plausible-but-wrong repairs that can introduce latent defects or security vulnerabilities; propagation of license-incompatible or proprietary snippets via retrieval; over-reliance on non-deterministic systems in safety- or mission-critical settings; and increased energy use from large models. Agentic systems can also widen the attack surface (e.g., prompt or retrieval injection) and may amplify biases present in code corpora.

Mitigations we recommend: enforce verification gates (compilation, unit and differential tests, fuzzing, and lightweight formal checks where feasible) before deployment; use privacy-preserving, in-tenant retrieval with provenance and license scanning; prefer conditional tool invocation and short prompts to reduce context leakage; log all tool calls and code diffs for audit; restrict side-effecting tools and require explicit human approval for high-impact actions; document stochasticity and provide reproducible decoding settings for review. The results here should be viewed as improving reliability in early pipeline stages, not as a substitute for semantic validation or secure development practices.

References

[1] Anthropic. Introducing the model context protocol, 2024.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Tobin, Jakub Pachocki, Aitor Ormazabal, Bob McGrew, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[3] Michael Feil. Infinity - to embeddings and beyond, oct 2023.

[4] Patrick Lewis et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *arXiv:2005.11401*, 2020.

[5] Qwen Team. Qwen3-235b-a22b. HuggingFace model card, 2025.

[6] Qwen Team. Qwen3-30b-a3b. `https://huggingface.co/Qwen/Qwen3-30B-A3B`, 2025. Accessed 2025-08-27.

[7] Qwen Team. Qwen3-8b. `https://huggingface.co/Qwen/Qwen3-8B`, 2025. Accessed 2025-08-27.

[8] Timo Schick et al. Toolformer: Language models can teach themselves to use tools. In *NeurIPS*, 2023.

[9] Joshua Schwartz, Matthew J. Kusner, and T. J. O'Donnell. Fine-tuning a foundational llm for vhdl code generation. *arXiv preprint arXiv:2405.09610*, 2024.

[10] Shailja Thakur et al. Verigen: A large language model for verilog code generation. *arXiv:2308.00708*, 2023.

[11] Prashanth Vijayaraghavan et al. VHDL-Eval: A framework for evaluating large language models in vhdl code generation. In *LAD 2024*, 2024.

[12] Shunyu Yao et al. React: Synergizing reasoning and acting in language models. In *ICLR*, 2023.

A Licenses and attributions

Models. Qwen3-8B, Qwen3-30B-A3B, Qwen3-235B-A22B. License: Apache License 2.0. Model cards hosted on Hugging Face specify Apache-2.0 for these checkpoints.

Embedding server. `michaelfeil/infinity`. License: MIT. Used to serve text embeddings.

Vector index. FAISS. License: MIT.

VHDL toolchain. GHDL. License: GNU GPL v2 or later (code); documentation under CC BY-SA. Used only as an external compiler/simulator; we do not redistribute GHDL.

MATLAB. Proprietary software by MathWorks, used under institutional license; governed by the MathWorks Software License Agreement (EULA). We do not redistribute MATLAB or MathWorks assets.

Hosting. Hugging Face Inference Endpoints used for model serving; usage governed by Hugging Face Terms of Service and Supplemental Terms for Inference Services.

Release. We do not release new datasets, models, or code in this work. All third-party assets were used within their respective licenses and terms.

B Reproducibility Details

Models: Qwen3-8B, Qwen3-30B-A3B, Qwen3-235B-A22B (default tokenizers). Decoding: temperature=0.6, top-p=1.0, top-k unset, max-new-tokens backend default (unset), no stop tokens. Trials: R=12 independent runs per function; K=3 candidates per run. Repair loop: max iterations T=10; stop on first syntax pass or when T is reached. Retrieval: Infinity embeddings; ℓ_2-normalize, FAISS IndexFlatIP (cosine), top-k=3; truncate retrieved examples to 1200 tokens keeping headers/imports/process scaffolding; retrieval is conditional on no-progress or GHDL errors matching `missing library/use/type/port/process`. Tooling: GHDL with `-std=08`; backend via Hugging Face endpoints or local Transformers $\geq$4.43 (insensitive under fixed decoding). Metrics: per-function candidate pass $p_i = \#\text{passes}/(R \times K)$; function pass, reach, final success in $\{0, 1\}$; report macro means over 42 functions.

B.1 Compute resources

Provider: Hugging Face Inference Endpoints. Backend: TGI (default image; version not recorded).

Qwen3-8B. Instance: 1× NVIDIA A100 80GB GPU; 11 vCPUs; 145 GB RAM. Autoscaling: min 0, max 1 replica; strategy "hardware usage"; idle scale-to-0 after 15 min. Concurrency: single replica. Runtime: total 4,933 min across 42 functions × R=12 trials (K=3, T=10), i.e., ≈82.2 GPU-hours. Per-trial wall-clock (one function, K=3, T=10): ≈9.79 min. Tokens: no explicit max-token limits set.

Qwen3-30B. Instance: 1× NVIDIA H200 141GB GPU; 23 vCPUs; 256 GB RAM. Autoscaling: min 0, max 1 replica. Concurrency: single replica. Runtime: total 5,388 min across 42 functions × R=12 trials (K=3, T=10), i.e., ≈89.8 GPU-hours. Per-function trial wall-clock (K=3, T=10): ≈10.69 min. Tokens: no explicit max-token limits configured.

Qwen3-235B. Environment: on-prem DGX (local). Backend: Transformers/TGI (version not recorded). Hardware: not logged. Replication guidance:

- FP16: $\geq$8× 80 GB GPUs (e.g., 8×A100 80GB) with tensor parallelism $\geq$8.
- INT8/4-bit AWQ: $\geq$4× 80 GB GPUs (e.g., 4×A100 80GB) with tensor parallelism $\geq$4.

- System RAM $\geq$512 GB; fast local storage for model weights; recent CUDA driver.

Autoscaling/concurrency: not applicable (single local server). Runtime and wall-clock: not logged; replication may expect slower throughput than 30B under the same decoding settings.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Yes. The abstract and introduction bound scope to a 42-function MATLAB-to-HDL dataset and the isolated syntax-repair stage, quantify key effects (e.g., 30B simulation reach 72.1%→95.3%, +23 pp), and note the mixed 235B outcome. Design attributions are framed as hypotheses and limitations are stated, so claims match the demonstrated results and their expected generality.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Yes. The paper includes a dedicated "Limitations and Future Work" section that bounds scope to a 42-function MATLAB→HDL dataset and the isolated syntax-repair stage, notes scale dependence, and states that downstream semantic correctness remains a bottleneck with limited generalization beyond signal-processing code. It also discloses reliance on an internal VHDL corpus and the absence of causal ablations, which are proposed as future work.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best

judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: The paper is empirical only and presents no theorems or formal claims requiring assumptions or proofs.

   Guidelines:

   - The answer NA means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We specify decoding and loop settings (t=0.6, top-p=1.0, K=3, R=12, T=10), retrieval and FAISS configuration, tooling, metric definitions, enabling independent verification within expected stochastic variation.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.

(c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: The dataset and internal tool flow are proprietary and cannot be released, and we do not include runnable code or scripts in the supplement. We provide detailed settings and a surrogate public replication recipe, but this is not open access to the original data or code.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: This work is inference-only (no training), and it specifies the evaluation setup: dataset (42 MATLAB functions, 3–311 LoC), pipeline/tools (GHDL –std=08), decoding and loop settings (t=0.6, top-p=1.0, K=3, R=12, T=10), retrieval config (Infinity embeddings, FAISS IndexFlatIP, top-k=3, truncation), and metric definitions. These details are sufficient to understand and interpret the reported results.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: We report macro averages only. We did not retain run-level outcomes, so we cannot compute confidence intervals. Future work will log full run traces for statistical reporting.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
   - If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Compute resources specifies provider/backend and, for 8B and 30B, the exact instance types (A100 80GB; H200 141GB), vCPU/RAM, autoscaling policy, per-function wall-clock, and total GPU-hours ($\approx$82.2h and $\approx$89.8h). For 235B it documents a local DGX environment and provides minimum hardware to reproduce (e.g., $\geq$8×80GB FP16 or $\geq$4×80GB INT8/4-bit) while noting runtime was not logged. This information is sufficient to size hardware and reproduce the experiments.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

Justification: The work uses company-authorized, non-personal code data only; no human subjects, PII, demographics, or scraping are involved. We respect model and hosting licenses (Qwen, Hugging Face), preserve anonymity, disclose limitations and compute usage, and present no foreseeable dual-use or safety risks beyond standard code synthesis.

Guidelines:

- The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: The Broader Impacts section discusses positive effects (productivity, lowered barriers, safer agentic workflows) and negatives (latent defects/security risks, IP leakage via retrieval, attack surface, energy use), and outlines mitigations (verification gates, provenance/license scanning, conditional tool use, logging/audit, restricted side-effects).

Guidelines:

- The answer NA means that there is no societal impact of the work performed.
- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

Answer: [NA]

Justification: We do not release models, checkpoints, or datasets. Experiments use publicly available Qwen checkpoints via Hugging Face; the internal corpus remains private. Only configuration details and minimal prompts are disclosed, so no high-risk assets require safeguards.

Guidelines:

- The answer NA means that the paper poses no such risks.

- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Appendix A credits all third-party assets and states licenses/terms (Qwen3 Apache-2.0; Infinity MIT; FAISS MIT; GHDL GPLv2+; MATLAB EULA; Hugging Face ToS). Usage complies with those terms; no proprietary assets are redistributed.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [NA]

    Justification: We do not release new datasets, models, or runnable code. The internal dataset and tooling are not shared. The paper includes inline prompts/tool schemas only as documentation, not as a released asset package.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [NA]

Justification: The work involves no crowdsourcing and no human-subject research.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [NA]

Justification: No human-subjects research or crowdsourcing was conducted, so no participant risks or IRB approvals apply.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary greatly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: LLMs are the core method: we use Qwen3-8B/30B/235B with specified decoding, trials, retrieval, and MCP tooling, all documented in the paper. Separately, an LLM assisted with minor copyediting only; it contributed no ideas, data, code, or analysis.

Guidelines:

- The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.