Small, Fast, and Certain: Developing a Specialized Verilog Code Completion Solution for the Enterprise

Eran Avidan[*]
Intel AI Solutions Group
`eran.avidan@intel.com`

Lior Tondovsky[*]
Intel AI Solutions Group
`lior.tondovsky@intel.com`

Amitai Armon
Intel AI Solutions Group
`amitai.armon@intel.com`

Abstract

We describe the development of a specialized code-completion solution for hardware designers in our company. It handles their specific flavor of System Verilog, and uses a low-latency on-prem fine-tuned model. We outline the process of developing this solution, from data curation, through several stages of model fine-tuning with different contexts, to evaluation and real-time confidence assessment. We then present our results for fine-tuning a 1B-parameter model on $\sim$1B tokens of in-domain System Verilog code, achieving high semantic fidelity and low latency for both end-of-line and multi-line completions. Our results demonstrate that small, specialized models can satisfy the latency and privacy requirements of enterprise deployment, offering a viable alternative to general-purpose LLMs in constrained settings.

1 Introduction

Code completion tools assist developers by suggesting relevant continuations, helping to boost productivity and reduce cognitive load [10]. However, applying these tools in hardware design environments presents unique challenges [8, 12, 4]. Hardware description languages (HDLs) such as Verilog and System Verilog feature specialized syntax and semantics that differ significantly from general-purpose programming languages. Enterprise teams also rely on internal conventions and proprietary terminology, making it difficult for general-purpose models to provide relevant and trustworthy completions.

While large language models (LLMs) perform well in broad settings, they often fall short in enterprise use cases due to latency constraints, lack of domain adaptation, and intellectual property concerns. This highlights the need for small, domain-specific models that run efficiently on local infrastructure and align with internal codebases.

We developed a System Verilog-specific code completion model, tailored for an enterprise specializing in HDL development. Inspired by TinyStories [3], which demonstrated the surprising effectiveness of small language models when trained on high-quality, domain-aligned data, we pursue a similar approach for our hardware design use-case. Our solution supports fast iteration, with low latency end-of-line and multi-line completions, and it can generalize to other domains. It includes a confidence-based truncation for coverage-versus-accuracy control, inspired by prior work on confidence-aware decoding in code models [11]. Our benchmarking approach mimics real System Verilog code completions drawn from the same dataset of production repositories, and assesses their correctness based on the ground truth. As a reference, we evaluated the latest OpenAI model with a completion API, GPT-3.5-turbo [9], as well as Mistral-7B [5]. While neither met our latency requirements, they served as baselines for comparison. Later models, such as GPT-4 and GPT-4o, could not be used for completion and are also significantly slower.

[*]Equal contribution.

Our main contributions are: **(1)** a practical procedure for fine-tuning domain-specific code completion models under enterprise constraints, **(2)** a simple confidence-based truncation method, enabling dynamic trade-offs between accuracy, coverage, and latency, and **(3)** a simple benchmark methodology, grounded in production usage scenarios, enabling reproducible and actionable evaluation, along with the results we obtained for our real use-case.

2 Methods

We present our end-to-end procedure for fine-tuning and benchmarking small transformer-based language models (SLMs) for low-latency code completion. Our process was designed to be adaptable across domains, programming languages, and deployment environments.

2.1 Data Curation

Data curation is clearly the foundation of domain adaptation. Our goal was to construct a corpus that closely reflects the target deployment domain in both structure and style. We began by collecting a large set of System Verilog source files from validated internal repositories. To ensure high-quality learning and prevent overfitting, we applied a standard multi-stage deduplication process. First, we removed exact duplicates by stripping comments and comparing the normalized content. Then, we identified near-duplicates using SHA1 hashes over subsets of the normalized files to detect near similar entries. Following this, we filtered out irrelevant artifacts such as autogenerated files (that had multiple duplicates), empty or trivial modules, and documentation-only blocks, resulting in a clean, domain-aligned dataset.

2.2 Model Preparation

Base Model Selection We targeted compact decoder-only models that were trained on code, aiming to satisfy a strict latency requirement of less than half a second for a multi-line completion. We selected two StarCoder models [7] with 1B parameters (StarCoderBase-1B) [2] and 164M parameters (TinyStarCoderPy) [3].

Sliding-Window Batching To increase training efficiency and expose the model to long-range dependencies, we applied overlapping sliding windows across source files. This strategy allows the model to see inter-block patterns that exceed the context length, improving generalization without increasing model size.

Three-Stage Fine-Tuning Fine-tuning proceeded in three stages: **(1) Domain Adaptation:** We began with standard next-token prediction on the curated domain data to align the model with local syntax, naming conventions, and structural patterns. **(2) Context Length Scaling:** We increased the context length—from 1k to 2k tokens—to enhance the model's ability to capture long-range dependencies. **(3) Fill-in-the-Middle (FIM) Training:** Following Bavarian et al. [1], we restructured each training example into a triplet of `<prefix>`, `<suffix>`, and `<middle>` segments. The model was trained to generate the middle span conditioned on both the prefix and suffix, improving its performance on insertion tasks such as middle-of-line or mid-function completions.

2.3 Evaluation Protocol

For the evaluation we sampled valid cursor positions from each test file, as described in Section 3.2. A single cursor per file may suffice for a large and diverse dataset with thousands of files of similar lengths. Sampling multiple cursors may improve robustness and increase context variety when the number of files is relatively small, or they have very different lengths. For each sampled cursor, the model was tasked with generating completions using the available prefix and suffix context. The primary evaluation metric was a whitespace-insensitive exact match (EM) between the predicted and ground-truth span. This was computed separately for end-of-line (EOL) and multi-line completions.

To account for cases where exact match (EM) fails despite semantically correct completions, we used a secondary metric based on functional or semantic equivalence, scored by an external LLM judge (GPT4). We identified at the beginning of our work that EM is highly correlated with the semantic score on our validation set, enabling us to adopt EM as the primary evaluation metric in

Table 1: EOL Exact match (EOL-EM) accuracy on the 10K test set for different model sizes and context lengths. Increasing the context from 1024 to 2048 led to clear gains in our fine-tuned StarCoder models (FT). For Mistral and GPT-3.5, larger context showed no significant benefit. FIM (fill-in-the-middle) was applied only to 2048-token models.

Model	Context Size	EOL-EM	Equivalence
StarCoder-1B-FT	1024	**65%**	**72%**
TinyStarCoder-164M-FT	1024	61%	68%
Mistral-7B	1024	48%	56%
GPT-3.5-turbo	1024	48%	55%
StarCoder-1B-FIM-FT	2048	**74%**	–
TinyStarCoder-164M-FIM-FT	2048	69%	–
Mistral-7B	2048	48%	–
GPT-3.5-turbo	2048	50%	–

subsequent stages. This choice made the evaluation process fully deterministic, more scalable, and less dependent on external models.

2.4 Confidence-Based Gating

To dynamically control completion length during inference, we used a simple confidence-based truncation method, using statistics computed over the entire generated sequence. We evaluated several heuristics—including average, median, minimum, and standard deviation of token probabilities. We found that **minimum token probability and standard deviation** exhibited the strongest correlation with exact match accuracy (point-biserial correlation [6] $r \approx 0.65$), while showing no correlation with completion length or context size. This suggested that these metrics are orthogonal to prompt characteristics and can serve as reliable, length-agnostic confidence signals. For deployment, we used the **minimum token probability** as a simple and robust stopping criterion, allowing us to truncate low-confidence completions dynamically, as soon as the probability of a token falls below our threshold. We evaluated several thresholds on held-out data, to enable flexible trade-offs: **Lower thresholds** yield longer completions and higher coverage. **Higher thresholds** produce shorter, more accurate completions. Users can tune the threshold to match latency and quality constraints.

3 Experiments

3.1 Setup

Data We constructed a high-quality dataset consisting of 100K real-world System Verilog files, sourced from production code repositories. The data was split into 80K files for training, 10K for validation, and 10K for testing, totaling approximately 1B tokens in the training set. To increase sample diversity and expose the model to more long-range patterns, we applied overlapping sliding windows with stride $= w/2$ (for each context window size mentioned below), effectively doubling the training tokens to approximately 2B per epoch.

Model We fine-tuned decoder-only StarCoder models [7] with 1B parameters (StarCoderBase-1B) [2] and 164M parameters (TinyStarCoderPy) [3], using the above three-stage fine-tuning process. The Tiny model had no SystemVerilog exposure, having been pretrained solely on Python. The 1B model had limited exposure, serving as a starting point for further fine-tuning for our own System Verilog variant. First, we trained them on next-token prediction on the curated dataset with a 1024-token context window to align the model with domain-specific patterns. Next, we continued training with a 2048-token context to enhance the model's ability to capture long-range dependencies. Finally, we applied FIM training to support insertion-style completions, structuring each example into prefix, middle, and suffix segments.

Table 2: Effect of probability threshold (τ) on coverage, accuracy, and median length statistics

Threshold	Coverage	EM	Med Tokens	Med Lines
0.85	86.40%	82.83%	13	2
0.80	88.20%	79.70%	16	2
0.75	89.70%	76.30%	18	2
0.70	91.00%	72.90%	21	2

3.2 Evaluation Protocol

Evaluation was conducted after each training phase using the 10K files test set, sampling at random one cursor positions per file. To assure evaluation quality, we excluded completions that consisted solely of comments and filtered out samples where the target line, along with its preceding line, had appeared in the training data (thus eliminating memorized completions). As expected, this filtering slightly reduced our accuracy measurement but provided more reliable results. Completions were assessed using two metrics: **Exact Match (EM)**, which checks for token-level equivalence (ignoring whitespace), and **Semantic Equivalence**, scored by a GPT-4-based judge to capture logical correctness. As mentioned above, after the first training phase, we observed strong correlations between EM and semantic correctness, indicating that EM is a reliable proxy for semantic quality. As a result, we used EM as the sole evaluation metric in later stages, simplifying the process while preserving reliability and reproducibility.

3.3 Results

Table 1 summarizes the EOL exact match (EOL-EM) accuracy on the 10K test set across different model sizes and context lengths. Our fine-tuned StarCoder models outperformed much larger general-purpose baselines. The 1B model improved from 65% EOL-EM at 1024 tokens to 74% at 2048 tokens, and the 164M model rose from 52% to 69%. These gains highlight the benefits of domain adaptation, long-context training, and the use of fill-in-the-middle (FIM) formatting, which was applied to the 2048-token variants. In contrast, Mistral-7B and GPT-3.5-turbo showed no significant improvement from increased context size and underperformed both StarCoder variants despite being significantly larger. Since extending StarCoder to 4096 tokens yielded minimal additional gains while increasing inference cost, we selected the StarCoder-1B-FIM-FT with 2048 context window configuration for deployment and confidence-based gating. Overall, the fine-tuning process improved this model's EOL-EM accuracy by 26% compared to our initial check with the base StarCoder-1B.

3.4 Confidence-Based Gating

To reduce latency and suppress low-confidence generations, we applied post-hoc truncation based on the token probability, as described above. We evaluated thresholds $\tau \in \{0.85, 0.80, 0.75, 0.70\}$ on the validation set of the StarCoder-1B-FIM-FT model with a 2048-token context. Table 2 summarizes how varying the minimum token probability threshold (τ) influences coverage (i.e., the proportion of completions retained), exact match (EM) accuracy, and the median completion length. Higher thresholds increase EM while reducing coverage.

4 Conclusion

We presented a practical procedure for training and deploying domain-specific code completion models in HDL environments. By combining scalable data curation, compact transformer fine-tuning, context scaling, and confidence-based truncation, our pipeline achieved high accuracy and low latency on real-world System Verilog code. The use of 2048-token context windows and FIM training significantly boosted model performance, while our post-hoc gating mechanism enabled real-time control over both coverage and accuracy—allowing a trade off without retraining. The entire pipeline—from dataset preparation to deployment—can be easily completed within a few days, enabling rapid iteration. We believe this procedure illustrates how fast, privacy-preserving, and domain-adapted code intelligent systems can be deployed entirely on-premise.

References

[1] Mohammad Bavarian, Aitor Lewkowycz, Barret Zoph, James Lee-Thorp, Noam Shazeer, and Yi Chen. Efficient training of language models to fill in the middle. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

[2] BigCode. starcoderbase-1b. `https://huggingface.co/bigcode/starcoderbase-1b`, 2023. Accessed: 2025-08-17.

[3] BigCode. tiny_starcoder_py. `https://huggingface.co/bigcode/tiny_starcoder_py`, 2023. Accessed: 2025-08-17.

[4] Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. Verilogcoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 300–307, 2025.

[5] Zihao Jiang et al. Mistral 7b, 2023. `https://mistral.ai/news/announcing-mistral-7b`.

[6] Diana Kornbrot. Pointbiserial correlation. In *Encyclopedia of Statistics in Behavioral Science*. Wiley, 2014.

[7] Raymond Li, Lewis Tunstall, Sasha Luccioni, Aleksandra Piktus, Younes Belkada, Swabha Swayamdipta, Albert Webson, Alexander Glaese, Teven Le Scao, Sharmila Chinchilla, et al. Starcoder: May the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[8] Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[9] OpenAI. Openai gpt-3.5 api documentation, 2023. `https://platform.openai.com/docs/models/gpt-3-5`.

[10] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of ai on developer productivity: Evidence from github copilot, 2023. URL `https://arxiv.org/abs/2302.06590`.

[11] Jiao Sun, Q Vera Liao, Michael Muller, Mayank Agarwal, Stephanie Houde, Kartik Talamadupula, and Justin D Weisz. Investigating explainability of generative ai for code through scenario-based design. In *Proceedings of the 27th international conference on intelligent user interfaces*, pages 212–228, 2022.

[12] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems*, 29(3):1–31, 2024.