VeriLoC: Line-of-Code Level Prediction of Hardware Design Quality from Verilog Code

Raghu Vamshi Hemadri[1*] **Jitendra Bhandari**[1*] **Andre Nakkab**[1] **Johann Knechtel**[2]
Badri P Gopalan[3] **Ramesh Narayanaswamy**[3] **Ramesh Karri**[1] **Siddharth Garg**[1]

[1]New York University Tandon School of Engineering
[2]New York University Abu Dhabi
[3]Synopsys

Abstract

Modern chip design is complex, and there is a crucial need for early-stage prediction of key design-quality metrics like timing and routing congestion directly from Verilog code (a commonly used programming language for hardware design). It is especially important yet complex to predict individual lines of code that cause timing violations or downstream routing congestion. Prior works have tried approaches like converting Verilog into an intermediate graph representation and using LLM embeddings alongside other features to predict module-level quality, but did not consider line-level quality prediction. We propose `VeriLoC`, the first method that predicts design quality directly from Verilog at both the line- and module-level. To this end, `VeriLoC` leverages recent Verilog code-generation LLMs to extract local line-level and module-level embeddings, and trains downstream classifiers/regressors on concatenations of these embeddings. `VeriLoC` achieves high F1-scores of 0.86–0.95 for line-level congestion and timing prediction, and reduces the mean average percentage error from $14\% - 18\%$ for SOTA methods down to only 4%. We believe that `VeriLoC` embeddings and insights from our work will also be of value for other predictive and optimization tasks for complex hardware design.

1 Introduction

Modern chip design is highly *complex*. It begins with devising a description of the chip's behavior in a hardware description language (HDL) like Verilog.[2] This is followed by a series of automated steps, including synthesis (where RTL code is converted into a circuit of Boolean logic and its gate implementation), placement (which arranges gates on the chip canvas), and routing (which connects gates using metal wires). This process transforms the RTL code into a manufacturable chip layout.

Key metrics for design quality, like area, timing, power, routing congestion, *etc.*, can only be verified from final layouts, but obtaining these layouts can take hours or days as synthesis, placement, routing, and other steps in the design flow are extremely complex and time-consuming. Designers often iterate multiple times till specifications and quality targets are met; these iterations can take anywhere from weeks to months, impacting time-to-market. Timing and routing congestion, in particular, are difficult to manage and are frequently the main impediment to design closure [1–5].

[*]Both authors contributed equally to this research.

[2]HDL codes are commonly also referred to as register-transfer level (RTL) descriptions. We use RTL or Verilog interchangeably from here on.

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

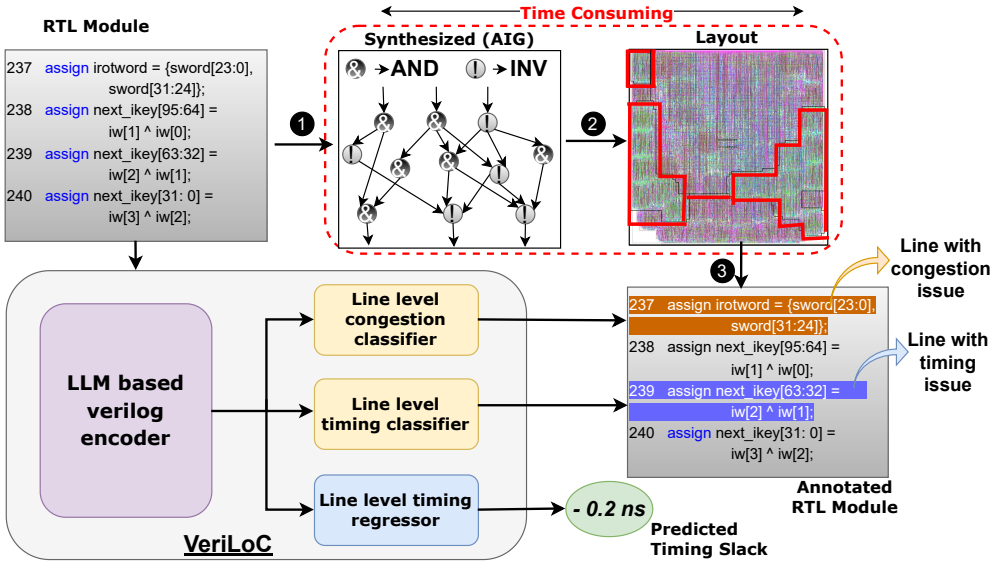

Figure 1: Conventional flow vs proposed for an exemplary AES design. ❶ The RTL code is converted into a synthesized netlist, e.g., represented by an AIG. ❷ The netlist is converted into a layout, with congestion arising in green areas (bounded in red). ❸ Congestion information is annotated and traced back to the RTL. With `VeriLoC`, we directly predict congestion and timing for the RTL at run-time, bypassing the time-consuming conventional steps.

To address this issue, a body of recent work has proposed *early-stage prediction* of design quality from RTL code, typically via intermediate representations of RTL like and-inverter graphs (AIGs)[3] [6–10]. However, intermediate representations might lose rich semantic information available in the RTL code in compact form. For example, a 64-bit multiplier is a single line of Verilog, but corresponds to hundreds or thousands of gates in an AIG, where information must then be *reverse engineered* by the ML model. Recent work has leveraged large language model (LLM) based encodings of Verilog modules for accurate module-level power, performance, and area (PPA) prediction [11, 12].

Aside from early-stage module-level predictions, designers can greatly from identifying *individual lines of code* (LoC) responsible for inducing timing violations or routing congestion. While electronic design automation (EDA) tools like RTL Architect [13] provide the capability of back-annotating the lines of code from a final layout (Step 3 in Fig. 1), these too are complex and time-consuming. Here, we pose a new research question that has not been addressed in literature: *can we predict design quality, specifically, timing and routing congestion, from RTL code at the module and the individual line-of-code level?*

A key challenge in addressing this question is how to obtain informative RTL embeddings—here, we leverage the recent emergence of LLMs trained specifically for Verilog code generation like *CL-Verilog* 13B [14]. Although *CL-Verilog* is a decoder-only model, recent studies [15, 16] have demonstrated that internal activations from these models can yield effective embeddings. Using penultimate layer outputs from *CL-Verilog* as embeddings, we propose `VeriLoC`, a novel architecture for line-level classification of Verilog code. To the best of our knowledge, the LoC-level prediction problem has not been addressed in literature before.

The key idea in the proposed `VeriLoC` architecture (Figure 2) is to concatenate embeddings of each line-of-code in a Verilog module with an embedding of the entire module, thus obtaining both a local and global context. In practice, we find that additionally concatenating embeddings from up to two *neighboring* lines further improves performance. `VeriLoC` then trains a supervised classifier (or regressor) on ground-truth data from Synopsys RTL Architect [17] on the OpenABCD dataset [18],

[3] An AIG is a Boolean circuit, which consists only of the so-called universal set of AND and NOT gates, and is at the same time functionally identical to the RTL.

using models like XGBoost [19] and LightGBM [20] tailored for scarce and imbalanced data. Our contributions are as follows:

- We propose and evaluate `VeriLoC`, a novel LLM-based architecture for early-stage prediction of hardware design quality *directly* from RTL code, both at the individual lines of code and entire modules. Prior work only performs module-level predictions and converts RTL to an intermediate representation, thereby losing rich semantic information.
- We identify the importance of capturing both the local context, i.e., neighboring lines of code, and global context, i.e., the entire Verilog module, in enabling line-level timing and congestion predictions. `VeriLoC`'s architecture concatenates both local and global embeddings before the final classification step.
- `VeriLoC` achieves F1-scores of 0.86 in line-level congestion prediction, 0.95 in line-level timing prediction, and also outperforms state-of-art in module-level timing prediction, reducing the mean average percentage error from 18% and 14% to only 4%.
- We demonstrate the usefulness of LLMs specialized for RTL code generation to also generate powerful RTL code *embeddings* that can be used for challenging downstream prediction tasks, specifically, timing and routing congestion prediction.

Overall, `VeriLoC`[4] establishes an entirely new approach for early-stage prediction from RTL code, which might be of value not only to other prediction tasks, but also for code and design optimization.

2 Background and Related Work

We discuss relevant background on hardware design and contrast `VeriLoC` with related work on predicting design quality and on LLM-based prediction of code quality.

2.1 Hardware Design: Quality Metrics and Prediction

2.1.1 Routing Congestion

What is Routing Congestion? Routing is one of the most complex and time-consuming steps in hardware design. Routing entails interconnecting logic gates with wires after the gates are placed on the chip canvas. Modern chips have tens of different routing layers, where any two wires that need to cross without connecting electrically can be routed above another, akin to a flyover in a traffic network. Almost every problem related to routing is known to be intractable [21]. Thus, like most processes in EDA, routing is heavily reliant on heuristic optimization, which cannot guarantee best quality in one go. In this context, managing *routing congestion*—or congestion for short—is important. This arises when multiple wires pass through the same small area of a chip, such that, in the worst case, the number of routing layers is insufficient to route all wires correctly, i.e., without at least two wires crossing paths. When this happens, the entire design might need to be undergo placement again, or might even necessitate an RTL rewrite.

Predicting Routing Congestion. Traditional methods commonly integrate actual routing processes [22–25] or analytical models that estimate congestion [26–29]. However, routing-based methods are plagued by considerable runtime cost while analytical-based approaches suffer from relatively low accuracy. To address these challenges, more recent works have employed ML techniques. For example, [30] utilize convolutional neural networks (CNNs) to predict the overall routability of placement solutions. In a follow-up work, [31] employ deep neural networks (DNNs) to achieve better performance for congestion prediction and guide toward less-congested placement. Furthermore, [32–37] all use graph neural networks (GNNs) using synthesized and/or placed netlists as inputs, which require running time-consuming synthesis and placement tools, respectively.

2.1.2 Timing

What is Timing? Timing is a critical aspect of chip design and determines the fastest frequency at which the chip can operate. Timing is affected by every process in the design cycle, but the most critical impact is within the RTL stage, as this dictates the architecture and data flow of the IC. Roughly, timing refers to the time it takes for data to propagate from a circuit's input to its output;

[4] https://github.com/ML4EDA/VeriLoC.git

thus, the longest sequence of gates from the input to the output is referred to as the critical path. Often, timing is measured by *worst negative slack (WNS)*, i.e., the difference/slack between the desired critical-path delay and the actual critical-path delay. The goal for designers is to push WNS above zero, i.e., to keep delays within the desired budget.

Timing Prediction. Prior works predict timing by employing various ML techniques at various design stages. Closest to our work, [1–4] predict timing for entire modules at the RTL stage, but use either AIGs or other intermediate representations. Of these, [4, 1] are the current state-of-the-art methods. `VeriLoC` demonstrates substantial accuracy improvements compared to both. Other methods propose timing prediction at later stages in the design, including after synthesis [5] and after placement [38, 39]. All of these methods are focused on module-level timing prediction; `VeriLoC` is the first method to provide line-of-code level predictions of WNS.

2.2 LLMs for Code Generation and Quality Prediction

2.2.1 LLMs for Software Code Quality

LLMs have demonstrated significant potential for coding, with applications spanning bug detection, program synthesis, and performance optimization [40]. Models like CodeBERT, GraphCodeBERT, and CodeT5 effectively capture syntactic and semantic nuances in high-level programming languages, making them invaluable for tasks like code summarization, translation, and repair [41, 42]. The aforementioned LLMs excel at these generative tasks. Bug detection can be viewed as a line-level prediction task, and has been addressed via pattern matching using static analysis [43], enhanced with LLMs [44], or by using LLMs for test generation and fuzzing [45]. In most instances, given the massive amounts of open-source software and vulnerability datasets, these methods can leverage LLMs with careful prompt tuning, retrieval, and agentic frameworks. Indeed, state-of-art approaches like LLMSAN [46] utilize few-shot chain-of-thought prompting to extract structured data-flow paths for bug detection, but do not make any architectural modifications. Unfortunately, data is scarce in hardware, and concepts like routing congestion are barely mentioned. As we show later, prompting methods fail completely for line-level congestion and timing estimation.

2.2.2 LLMs for Hardware

While LLMs originally targeted software code, recent work has extended their use to hardware description languages such as Verilog. Generative Verilog models (e.g., VeriGen [47], CLVerilog [14], RTLCoder [48] and Others [49–52]) achieve impressive synthesis quality but do not provide downstream quality-of-results (QoR) metrics. Building on this trend, RTLRewriter applies LLM-guided rewriting for optimization [53], and RTLFixer employs LLM-driven debugging to correct syntax errors at scale [54]. Beyond generative tasks, LLMs have begun to assist QoR estimation for rapid design-space exploration. For PPA estimation, multimodal techniques—including CircuitFusion and VeriDistill hardware code with structural or graph-based embeddings to predict power, performance, and area [12, 11]. However, these methods operate at module or graph granularity, leaving line-level semantics unexplored. To the best of our knowledge, `VeriLoC` is the first ever line-level QoR predictor using a hardware-specialized LLM, enabling prediction of timing and congestion metrics directly at the statement level in Verilog code.

3 Methodology

Overview. `VeriLoC` builds on the premise that LLMs customized for RTL/Verilog code generation that have recently begun to emerge can be also be used as embeddings that capture the semantics of RTL code and used for downstream prediction tasks. `VeriLoC` is the first to demonstrate this property in the hardware context. We illustrate our methodology in Fig. 2. Our approach relies on embeddings generated by *CL-Verilog* [14], a variant of LLaMA-2 fine-tuned on Verilog code, extracted from its penultimate layer activations. We use these embeddings hierarchically, offering semantic representations at both the **module-** (Sec. 3.1) and **line-level** (Sec. 3.2), potentially with more context from neighboring lines (Sec. 3.2). These embeddings are projected to a lower dimension (Sec. 3.3), concatenated and a final classification/regression head outputs line- and module-level predictions (Sec. 3.5).

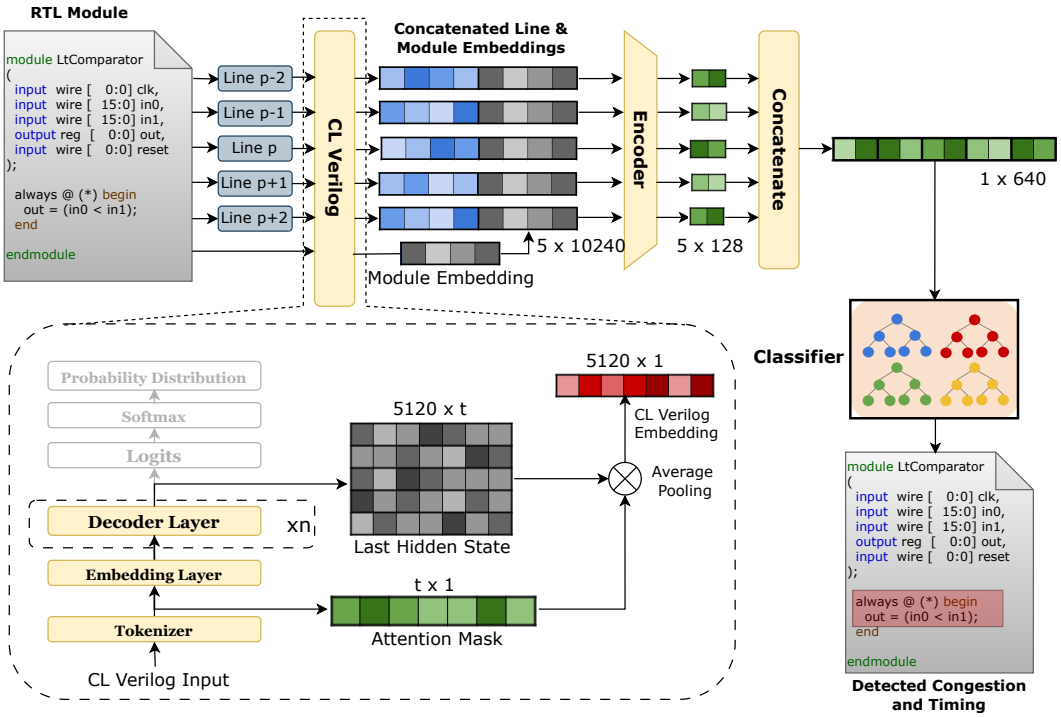

Figure 2: The architecture of `VeriLoC` for line-level timing and congestion prediction from RTL. Module-level prediction uses module embedding. The context window is set to $p = 5$ in this example.

3.1 Module-Level Embeddings

Modules in RTL designs are represented as sequences of lines of Verilog: $M = \{l_1, l_2, \ldots, l_n\}$, where l_i is the i-th line of code in the module. To capture the global semantics of the module, the complete module is passed through *CL-Verilog*, which generates hidden states $H = \{h_1, h_2, \ldots, h_n\}$, where $h_i \in \mathbb{R}^k$ is the latent vector for the i-th token of *CL-Verilog*'s tokenizer, and k is the dimensionality of the model's hidden state. Module embeddings $e(M)$ are computed as the pooled dot products of the hidden states and attention mask, normalized by sum of the attention mask:

$$e(M) = \frac{\sum_{i=1}^{n}(h_i \cdot m_i)}{\sum_{i=1}^{n} m_i}.$$

where $m_i \in \{0, 1\}$ is the attention mask that ensures only valid tokens contribute to the embedding, and the dot product $h_i \cdot m_i$ highlights the importance of each hidden state relative to the mask. The resulting module embedding $e(M) \in \mathbb{R}^k$ provides a condensed global representation of the module, enabling the detection of macro-level patterns such as resource utilization and timing violations.

3.2 Line-Level Embeddings

To capture localized semantics of Verilog and their impact on design quality, embeddings are also generated for individual lines of code. Each line l_i is passed independently through *CL-Verilog*, producing a hidden state h_i. Similar to module embeddings, the line embedding $e(l_i)$ is computed using the attention-weighted pooling mechanism:

$$e(l_i) = \frac{\sum_{i=1}^{n}(h_i \cdot m_i)}{\sum_{i=1}^{n} m_i}.$$

These embeddings $e(l_i) \in \mathbb{R}^k$ focus on the specific performance characteristics of each line, such as whether it contributes to congestion or WNS.

3.3 Dimensionality Reduction

After extracting module-level and line-level embeddings, dimensionality reduction is applied to ensure computational efficiency and improve downstream task performance. The combined embedding $x_i = [e(l_i); e(M)]$ undergoes dimensionality reduction as follows.[5] An encoder-decoder framework is trained to reconstruct the original concatenated embedding x_i from its reduced representation. The encoder maps the high-dimensional input x_i to a lower-dimensional space $\mathbb{R}^d$: $z_i = \textit{Encoder}(x_i)$, while the decoder reconstructs x_i from z_i: $\Phi(x_i) = \textit{Decoder}(\textit{Encoder}(x_i))$. The framework is optimized to minimize the reconstruction loss: $\mathcal{L} = \|x_i - \Phi(x_i)\|^2$. Once training is complete, only the encoder is retained for dimensionality reduction. In short, the encoder provides compact embeddings z_i and serves as a pre-trained initialization for downstream classification and regression.

3.4 Contextual Feature Augmentation

Contextual feature augmentation enhances the representation of a target line by integrating dependencies from surrounding lines. It captures sequential patterns and inter-line relationships, which are essential for analyzing RTL code. The embeddings of a target line l_i are concatenated with those of its neighboring lines within a context window p. For any l_i, the augmented embedding is: $z_{\text{aug}}(l_i) = [z_{i-p}; \ldots; z_i; \ldots; z_{i+p}]$, where z_i is the reduced embedding and $[\cdot]$ denotes vector concatenation. This approach introduces local dependencies, enabling the classifier to capture the sequential nature of RTL designs.

As shown in Fig. 3 (left), the statement `always @(posedge clk) begin` is not flagged, but in Fig. 3 (right), with `maybe_full <= N17;` introduced, the same line `always @(posedge clk) begin` becomes congestion-causing. This shows context-aware analysis can enhance detection by considering dependencies between neighbouring lines.

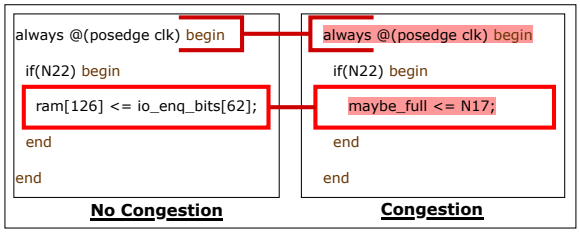

Figure 3: Effect of neighbor embeddings in context-aware congestion and timing detection.

3.5 Classification and Regression Heads

The concatenated embeddings feed into a classification head for line-level classification of code that cause congestion and timing issues, and a regression head for WNS prediction. We compare three classification/regression heads: (1) **Feedforward Neural Networks (FNNs)**, a single-layer and fully connected neural network that replaces *CL-Verilog*'s original classification head, but with a single classification (or regression) output; (2) **XGBoost**, a gradient-boosted tree [19]; (3) **LightGBM**, a lightweight gradient-boosting framework optimized for speed and performance [55]. XGBoost and LightGBM are used because of their demonstrated performance on imbalanced datasets [56]. This is essential for our work as only a small number of lines of code cause congestion or timing issues.

Although our primary focus is LoC-level prediction, we also use `VeriLoC` to estimate *module-level* WNS. Specifically, we we first estimate WNS at the line level and then select the worst (smallest) value across all lines. This line-wise granular prediction strategy allows the model to capture granular WNS estimates, and improves upon state-of-art module-level WNS predictors that use only module-level embeddings or features. This formulation, to the best of our knowledge, is unique to `VeriLoC`.

4 Empirical Evaluation

4.1 Experimental Setting

Dataset. We use the popular OpenABCD [18] RTL/Verilog code dataset for our experiments, using various Verilog modules from all projects in the dataset. We employed an 80/20 random split of the dataset to obtain training vs. test data. Dataset characteristics are shown in Table 1.

[5] Implementation and training is detailed in Appendix B.

To generate labels for timing and congestion, we use RTL Architect from Synopsys [13], transforming all designs to their physical layout, using the open-source *Nangate 45nm* standard-cell library. We ran all our designs with an aggressive timing constraint of `0.25` ns, because Synopsys RTL-A only reports WNS when a timing constraints are actually violated.

Table 1: Characteristics of OpenABCD and extracted Verilog. Designs have between 300–30K LOC.

Design	# of Modules	# of Lines	Design	# of Modules	# of Lines
`aes`	2	301	`coyote`	114	176279
`ariane`	39	214930	`dynamic_node`	9	796
`black_parrot`	88	62948	`ethmac`	10	1168
`bp_be_top`	38	13289	`jpeg`	5	669
`bp_fe_top`	15	7363	`microwatt`	31	26033
`bp_multi_top`	89	32959	`swerv_wrapper`	57	16496
`bp_quad`	252	293281	`vanilla5`	39	11577

Hyperparameter Setting. For congestion and timing detection, we employed XGBoost [19], LightGBM [20], and an FNN, each tuned to handle class imbalance and optimize predictive performance. XGBoost was configured with default hyperparameters: `scale_pos_weight` set as the ratio of the majority to minority class to mitigate imbalance, `max_depth=30`, `learning_rate=0.05` and `n_estimators=500`. LightGBM used `is_unbalance=True` for automatic class weight adjustment, and default settings of `num_leaves=100`, `learning_rate=0.05`, and `feature_fraction=0.8`. The regression head followed a similar training procedure using the 'XGBRegressor' with a squared error loss. The FNN consisted of a single sigmoid neuron trained with binary cross-entropy (BCE) loss and was optimized using Adam with a learning rate of $1e^{-4}$.

Metrics. For congestion and timing classification tasks, we use the **F1-score**, **precision**, and **recall** to measure the balance between sensitivity and specificity. For the regression task of WNS prediction, we employ $\mathbf{R^2}$ and **mean absolute percentage error (MAPE)**, providing insights for goodness-of-fit and prediction error relative to the target.

Hardware. *CL-Verilog* feature extraction was performed on a single NVidia H100, and downstream classifiers (XGBoost and LightGBM) were trained/evaluated on a CPU machine with 32GB RAM and 8 CPU cores. The FNN model was trained/evaluated using an NVidia RTX 8000 GPU.

4.2 Line-level Classification Results

We begin by discussing `VeriLoC`'s performance on line-level classification for both congestion and timing prediction. Table 2 tabulates our results for three different classification heads, as well as different context lengths ($p = \{1, 3, 5\}$).

Congestion Detection. As shown in Table 2, the highest F1-score for congestion detection is **0.86**, achieved by the LightGBM classifier with a context length of 5. This result underscores the importance of contextual information in accurately identifying congestion-causing lines.

Timing Detection. The highest F1-score of **0.95** is obtained using the XGBoost and LightGBM classifiers with a context length of 5, similar to the best performing model for congestion detection. Interestingly, timing prediction results are less sensitive to local context compared to congestion prediction. We achieve F1-scores of 0.83 for timing prediction even without local context ($p = 0$).

Table 2: Performance of `VeriLoC` on line-level congestion and timing detection. Best results are highlighted in blue.

Classifier	Context Length	Congestion			Timing		
		P	R	F1	P	R	F1
FNN	0	0.38	0.74	0.50	0.67	0.88	0.76
	3	0.41	0.77	0.54	0.71	0.89	0.80
	5	0.86	0.70	0.77	0.76	0.92	0.83
XGB	0	0.38	0.74	0.50	0.76	0.92	0.83
	3	0.42	0.78	0.55	0.91	0.93	0.92
	5	0.94	0.78	0.85	0.94	0.94	0.94
LGBM	0	0.38	0.78	0.51	0.82	0.91	0.83
	3	0.41	0.76	0.53	0.93	0.92	0.92
	5	0.94	0.79	0.86	0.96	0.94	0.95

These results reinforce prior observations about the advantage of XGBoost and LightGBM over deep networks in handling imbalanced datasets and irregular feature distributions [56], albeit these results were in the context of tabular data. XGBoost and LightGBM improve F1-scores from 0.77 to 0.86 for congestion prediction and 0.83 to 0.95 for timing prediction.

As shown in Table 2, they consistently outperform FNNs in both congestion and timing detection, particularly with a larger context of 5. The highest F1-scores for congestion detection (**0.86**) and timing detection (**0.95**) were achieved by these models, reinforcing their robustness in handling

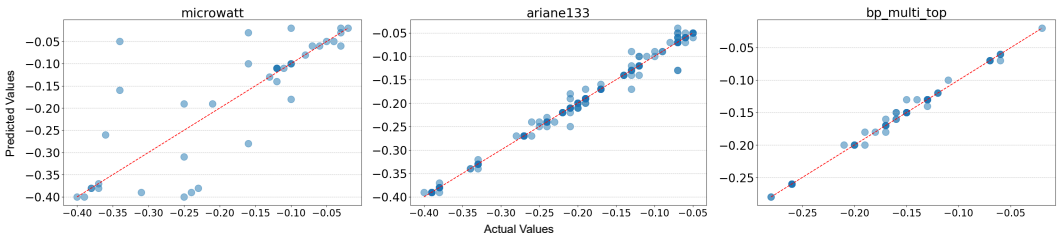

Figure 4: Scatter plots of actual vs. predicted line-level WNS using VeriLoC for three Verilog projects. In most instances, the predictions follow the actual WNS closely.

imbalanced datasets. The computational efficiency of XGBoost and its robustness in optimizing for minority class representation contributed to its superior performance compared to FNNs.

4.3 Timing Prediction and Comaprisons with SoTA

Table 3: Line-level prediction of WNS using VeriLoC. Module level prediction obtained from line-level ones.

Design	R^2	MAPE	Design	R^2	MAPE
aes	0.97	0.03	coyote	0.99	0.03
ariane	0.96	0.06	dynamic_node	0.76	0.18
black_parrot	0.99	0.01	ethmac	0.96	0.05
bp_be_top	0.76	0.27	jpeg	0.99	0.07
bp_fe_top	0.99	0.02	microwatt	0.93	0.21
bp_multi_top	0.99	0.02	swerv_wrapper	0.94	0.08
bp_quad	0.98	0.02	vanilla5	0.99	0.02

Table 3 reports VeriLoC's performance for predicting WNS at the line-level. Across all designs, VeriLoC's achieves high R^2 and low MAPE across all designs but two. The accuracy of our WNS predictions are also evident Figure 4, where we compare actual vs. predicted WNS for all lines in three benchmarks.

We now compare VeriLoC's module-level WNS prediction against state-of-art approaches. Recall that VeriLoC's module-level WNS predictions are obtained by *first* predicting WNS of each line in the code and then picking the smallest WNS, while other methods only use entire module-level features. Table 4 compares VeriLoC with MasterRTL [4] and RTL-Timer [1], SoTA methods that use handcrafted features derived from RTL. Across designs, VeriLoC outperforms prior art, with higher R^2 values (always $>$0.90) and much lower MAPE (always $<$0.10).

Table 4: VeriLoC vs. SOTA on module-level WNS (timing) prediction. VeriLoC substantially improves R^2 and MAPE. Best results are in blue.

Design	MasterRTL		RTL-Timer		VeriLoC	
	R^2	MAPE	R^2	MAPE	**R^2**	**MAPE**
aes	0.67	0.26	0.76	0.23	0.97	0.08
ariane	0.71	0.15	0.79	0.11	0.93	0.07
black_parrot	0.75	0.10	0.83	0.07	0.99	0.02
bp_be_top	0.76	0.12	0.87	0.08	0.98	0.08
bp_fe_top	0.80	0.09	0.88	0.05	0.98	0.04
bp_multi_top	0.61	0.16	0.67	0.13	0.99	0.02
bp_quad	0.69	0.14	0.78	0.10	0.94	0.06
coyote	0.74	0.15	0.83	0.12	0.99	0.03
dynamic_node	0.73	0.20	0.80	0.16	0.99	0.06
ethmac	0.77	0.24	0.83	0.21	0.99	0.03
jpeg	0.82	0.29	0.90	0.26	0.98	0.04
microwatt	0.81	0.20	0.80	0.16	0.99	0.02
swerv_wrapper	0.69	0.27	0.76	0.24	0.95	0.07
vanilla5	0.68	0.19	0.74	0.16	0.98	0.07

Table 5 compares VeriLoC with three additional baselines on module-level WNS: GNN-based predictors from synthesized netlists [10], VeriDistill [11] that uses LLM-based RTL embeddings with GNN-based synthesized look-up-table (LUT) embeddings, and as an ablation, VeriLoC-mod, a version of VeriLoC (VeriLoC-mod) that only uses module-level but no line-level embeddings. VeriLoC achieves the best performance, notably improving upon VeriLoC-mod, demonstrating the value of line-level embeddings even for module-level timing prediction. VeriDistill is second on MAPE, but performs poorly on R^2.

Direct comparisons of VeriLoC with *CircuitFusion* [12] method, a very recent, state-of-art, module-level multimodal PPA predictor, are challenging because they use a different dataset. Still, when comparing the respective improvements upon RTL-Timer, we see that *CircuitFusion* reports an increase in R^2 from 0.81 $\rightarrow$ 0.83 and a reduction in MAPE from 16% $\rightarrow$ 11%, whereas VeriLoC demonstrates larger relative gains, with R^2 increasing from 0.86 $\rightarrow$ 0.94, and MAPE reducing from 12% $\rightarrow$ 6%. We caution against reading more

Table 5: Comparison with SOTA methods for timing prediction on the OpenABCD benchmark. VeriLoC-mod uses *only* module embeddings but no line embeddings.

Metric	GNN [10]	MasterRTL [4]	RTL-Timer [1]	VeriDistill [11]	VeriLoC-Mod	VeriLoC
R^2	0.69	0.74	0.86	0.728	0.91	**0.94**
MAPE	0.17	0.15	0.12	0.076	0.08	**0.06**

into these comparisons because they are on different datasets, and also emphasize that VeriLoC's primary goal of LoC-level predictions is different from *CircuitFusion*'s.

4.4 Discussion

We now comment on some interesting properties of VeriLoC, avenues for future improvement, and alternate baselines.

Runtime Comparisons. We compare the runtime efficiency of VeriLoC over Synopsys RTL Architect [13] (our synthesis and PnR tool)—using *CL-Verilog*-13B as its base, VeriLoC achieves 14× average speedup and a median speedup of 61×. To explore the potential for even greater runtime improvement, we trained a 7B *CL-Verilog* model using the training procedure described in [14]. The 7B model has a 22× on average and 113× median speedup with a modest tradeoff in F1 score of 0.93 for timing and 0.84 for congestion. Thus, while the focus of VeriLoC is on accurate line-level PPA prediction, smaller models like the 7B variant or Mixture-of-Experts (MoE) based architectures can be adopted to further prioritize inference speed without significant degradation in predictive accuracy.

Impact of Verilog Code Length on Accuracy. To assess the effect of Verilog file length on model performance, we conducted a stratified analysis by splitting test samples based on the number of lines per file. We observed a negligible degradation in performance for longer inputs. Specifically, the **congestion F1-score** dropped slightly from **0.862** for files with fewer than 2000 lines to **0.855** for files with more than 2000 lines. Similarly, the **timing F1-score** decreased from **0.953** to **0.946**, and the **Mean Absolute Percentage Error (MAPE)** increased from **5%** to **6.5%**. These results indicate that the model exhibits strong robustness to variations in input length, with only minor performance degradation observed on longer files.

Can Black-Box LLMs Predict Hardware Design Quality? As we have noted before, much of the work in the software community on line-level bug detection uses prompt engineering with pre-trained models. These strategies have been successful because of the abundance of training data in the software world. However, hardware data is scarce, especially so for complex concepts like routing congestion. Thus, we hypothesize that prompting strategies are unlikely to work in the hardware context for line-level detection of congestion and timing issues. To test this hypothesis, we picked 10 Verilog files from our test dataset and asked ChatGPT-4o [57] to identify lines of code that would cause timing or congestion issues. ChatGPT was unable to identify lines responsible for congestion in *any* of our trials. For timing, ChatGPT identified a line of code correctly in one instance, but also had a large number of false positives. Examples of chat responses are given in Appendix D (Fig. 6–9).

5 Conclusion

This paper presents a novel LLM-driven framework, VeriLoC, which provides real-time feedback to hardware designers for the impact of their RTL code on crucial design-quality metrics, timing and congestion. By leveraging embedding from LLMs, VeriLoC bridges the gap between RTL coding and downstream performance evaluation, enabling designers to make informed decisions early in the design process. Results demonstrate VeriLoC's capability in predicting design metrics at RTL stage, both for individual lines of code and at the module level.

Limitations. Despite its strong performance, VeriLoC has several important limitations. First, it currently operates on leaf modules drawn from the Open-ABCD benchmark and has not been adapted to large-scale industrial designs with inter-module dependencies. Extensive validation on diverse, industry-grade RTL corpora will establish robustness and generalization, but is hindered by open access to such data.

References

[1] W. Fang, S. Liu, H. Zhang, and Z. Xie, "Annotating slack directly on your verilog: Fine-grained rtl timing evaluation for early optimization," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3649329.3655671

[2] D. S. Lopera and W. Ecker, "Applying gnns to timing estimation at rtl : (invited paper)," in *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2022, pp. 1–8.

[3] D. Sánchez Lopera, I. Subedi, and W. Ecker, "Using graph neural networks for timing estimations of rtl intermediate representations," in *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*, 2023, pp. 1–6.

[4] W. Fang, Y. Lu, S. Liu, Q. Zhang, C. Xu, L. W. Wills, H. Zhang, and Z. Xie, "Masterrtl: A pre-synthesis ppa estimation framework for any rtl design," in *Proceedings of 2023 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.

[5] Z. Xie, R. Liang, X. Xu, J. Hu, C.-C. Chang, J. Pan, and Y. Chen, "Preplacement net length and timing estimation by customized graph neural network," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4667–4680, 2022.

[6] A. Hosny, S. Hashemi, M. Shalan, and S. Reda, "Drills: Deep reinforcement learning for logic synthesis," *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 581–586, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:207853198

[7] K. Zhu, M. Liu, H. Chen, Z. Zhao, and D. Z. Pan, "Exploring logic optimizations with reinforcement learning and graph convolutional network," in *2020 ACM/IEEE 2nd Workshop on Machine Learning for CAD (MLCAD)*, 2020, pp. 145–150.

[8] Y. V. Peruvemba, S. Rai, K. Ahuja, and A. Kumar, "Rl-guided runtime-constrained heuristic exploration for logic synthesis," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–9.

[9] A. Basak Chowdhury, B. Tan, R. Carey, T. Jain, R. Karri, and S. Garg, "Bulls-eye: Active few-shot learning guided logic synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 8, pp. 2580–2590, 2023.

[10] A. B. Chowdhury, M. Romanelli, B. Tan, R. Karri, and S. Garg, "Retrieval-guided reinforcement learning for boolean circuit minimization," *arXiv preprint arXiv:2401.12205*, 2024.

[11] R. Moravej, S. Bodhe, Z. Zhang, D. Chetelat, D. Tsaras, Y. Zhang, H.-L. Zhen, J. Hao, and M. Yuan, "The graph's apprentice: Teaching an llm low level knowledge for circuit quality estimation," 2025. [Online]. Available: https://arxiv.org/abs/2411.00843

[12] W. Fang, S. Liu, J. Wang, and Z. Xie, "Circuitfusion: Multimodal circuit representation learning for agile chip design," in *International Conference on Learning Representations (ICLR)*, 2025. [Online]. Available: https://arxiv.org/abs/2505.02168

[13] "Synopsys rtl architect," https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/rtl-architect-ds.pdf.

[14] A. Nakkab, S. Q. Zhang, R. Karri, and S. Garg, "Rome was not built in a single step: Hierarchical prompting for llm-based chip design," in *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, ser. MLCAD '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3670474.3685964

[15] P. BehnamGhader, V. Adlakha, M. Mosbach, D. Bahdanau, N. Chapados, and S. Reddy, "Llm2vec: Large language models are secretly powerful text encoders," 2024. [Online]. Available: https://arxiv.org/abs/2404.05961

[16] Y. Luo, T. Zheng, Y. Mu, B. Li, Q. Zhang, Y. Gao, Z. Xu, P. Feng, X. Liu, T. Xiao, and J. Zhu, "Beyond decoder-only: Large language models can be good encoders for machine translation," 2025. [Online]. Available: https://arxiv.org/abs/2503.06594

[17] "Synopsys rtl architect," https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/rtl-architect-ds.pdf.

[18] A. B. Chowdhury, B. Tan, R. Karri, and S. Garg, "Openabc-d: A large-scale dataset for machine learning guided integrated circuit synthesis," 2021.

[19] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[20] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: a highly efficient gradient boosting decision tree," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 3149–3157.

[21] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu, *VLSI Physical Design: From Graph Partitioning to Timing Closure*. Springer, 2011.

[22] M.-C. Kim, J. Hu, D.-J. Lee, and I. L. Markov, "A simplr method for routability-driven placement," in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2011, pp. 67–73.

[23] X. He, T. Huang, W.-K. Chow, J. Kuang, K.-C. Lam, W. Cai, and E. F. Young, "Ripple 2.0: High quality routability-driven placement via global router integration," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–6.

[24] W.-H. Liu, C.-K. Koh, and Y.-L. Li, "Optimization of placement solutions for routability," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–9.

[25] C.-C. Huang, H.-Y. Lee, B.-Q. Lin, S.-W. Yang, C.-H. Chang, S.-T. Chen, Y.-W. Chang, T.-C. Chen, and I. Bustany, "Ntuplace4dr: A detailed-routing-driven placer for mixed-size circuit designs with technology and region constraints," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 3, pp. 669–681, 2018.

[26] P. Spindler and F. M. Johannes, "Fast and accurate routing demand estimation for efficient routability-driven placement," in *2007 Design, Automation & Test in Europe Conference & Exhibition*, 2007, pp. 1–6.

[27] Y. Wei, C. Sze, N. Viswanathan, Z. Li, C. J. Alpert, L. Reddy, A. D. Huber, G. E. Tellez, D. Keller, and S. S. Sapatnekar, "Glare: Global and local wiring aware routability evaluation," in *DAC Design Automation Conference 2012*, 2012, pp. 768–773.

[28] X. He, T. Huang, L. Xiao, H. Tian, and E. F. Y. Young, "Ripple: A robust and effective routability-driven placer," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 10, pp. 1546–1556, 2013.

[29] J.-M. Lin, C.-W. Huang, L.-C. Zane, M.-C. Tsai, C.-L. Lin, and C.-F. Tsai, "Routability-driven global placer target on removing global and local congestion for vlsi designs," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–8.

[30] Z. Xie, Y.-H. Huang, G.-Q. Fang, H. Ren, S.-Y. Fang, Y. Chen, and J. Hu, "Routenet: Routability prediction for mixed-size designs using convolutional neural network," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[31] S. Zheng, L. Zou, S. Liu, Y. Lin, B. Yu, and M. Wong, "Mitigating distribution shift for congestion optimization in global placement," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1–6.

[32] S. Yang, Z. Yang, D. Li, Y. Zhang, Z. Zhang, G. Song, and J. Hao, "Versatile multi-stage graph neural network for circuit representation," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 20 313–20 324.

[33] R. Kirby, S. Godil, R. Roy, and B. Catanzaro, "Congestionnet: Routing congestion prediction using deep graph neural networks," in *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, 2019, pp. 217–222.

[34] B. Wang, G. Shen, D. Li, J. Hao, W. Liu, Y. Huang, H. Wu, Y. Lin, G. Chen, and P. A. Heng, "Lhnn: lattice hypergraph neural network for vlsi congestion prediction," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1297–1302. [Online]. Available: https://doi.org/10.1145/3489517.3530675

[35] A. Ghose, V. Zhang, Y. Zhang, D. Li, W. Liu, and M. Coates, "Generalizable cross-graph embedding for gnn-based congestion prediction," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE Press, 2021, p. 1–9. [Online]. Available: https://doi.org/10.1109/ICCAD51958.2021.9643446

[36] S. Zheng, L. Zou, P. Xu, S. Liu, B. Yu, and M. Wong, "Lay-net: Grafting netlist knowledge on layout-based congestion prediction," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023, pp. 1–9.

[37] K. Baek, H. Park, S. Kim, K. Choi, and T. Kim, "Pin accessibility and routing congestion aware DRC hotspot prediction using graph neural network and u-net," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2022.

[38] Z. Guo, M. Liu, J. Gu, S. Zhang, D. Z. Pan, and Y. Lin, "A timing engine inspired graph neural network model for pre-routing slack prediction," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1207–1212. [Online]. Available: https://doi.org/10.1145/3489517.3530597

[39] P. Cao, G. He, and T. Yang, "Tf-predictor: Transformer-based prerouting path delay prediction framework," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 7, pp. 2227–2237, 2023.

[40] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[41] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139/

[42] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. LIU, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcode{bert}: Pre-training code representations with data flow," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=jLoC4ez43PZ

[43] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1646–1656. [Online]. Available: https://doi.org/10.1145/3611643.3613892

[44] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 474–499, 2024.

[45] K. Liu, Y. Liu, Z. Chen, J. M. Zhang, Y. Han, Y. Ma, G. Li, and G. Huang, "Llm-powered test case generation for detecting tricky bugs," *arXiv preprint arXiv:2404.10304*, 2024.

[46] C. Wang, W. Zhang, Z. Su, X. Xu, and X. Zhang, "Sanitizing large language models in bug detection with data-flow," in *Findings of the Association for Computational Linguistics: EMNLP 2024*, Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, Eds. Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 3790–3805. [Online]. Available: https://aclanthology.org/2024.findings-emnlp.217/

[47] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "Verigen: A large language model for verilog code generation," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 29, no. 3, Apr. 2024. [Online]. Available: https://doi.org/10.1145/3643681

[48] S. Liu, W. Fang, Y. Lu, Q. Zhang, H. Zhang, and Z. Xie, "Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution," in *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 2024, pp. 1–5.

[49] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "Verilogeval: Evaluating large language models for verilog code generation," 2023. [Online]. Available: https://arxiv.org/abs/2309.07544

[50] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "Rtllm: An open-source benchmark for design rtl generation with large language model," in *Proceedings of the 29th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '24. IEEE Press, 2024, p. 722–727. [Online]. Available: https://doi.org/10.1109/ASP-DAC58780.2024.10473904

[51] Y. Zhao, D. Huang, C. Li, P. Jin, M. Song, Y. Xu, Z. Nan, M. Gao, T. Ma, L. Qi, Y. Pan, Z. Zhang, R. Zhang, X. Zhang, Z. Du, Q. Guo, and X. Hu, "Codev: Empowering llms with hdl generation through multi-level summarization," 2025. [Online]. Available: https://arxiv.org/abs/2407.10424

[52] Z. Pei, H. Zhen, M. Yuan, Y. Huang, and B. Yu, "BetterV: Controlled verilog generation with discriminative guidance," in *Proceedings of the 41st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, R. Salakhutdinov, Z. Kolter, K. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp, Eds., vol. 235. PMLR, 21–27 Jul 2024, pp. 40 145–40 153. [Online]. Available: https://proceedings.mlr.press/v235/pei24e.html

[53] X. Yao, Y. Wang, X. Li, Y. Lian, R. Chen, L. Chen, M. Yuan, H. Xu, and B. Yu, "Rtlrewriter: Methodologies for large models aided rtl code optimization," 2024. [Online]. Available: https://arxiv.org/abs/2409.11414

[54] Y.-D. Tsai, M. Liu, and H. Ren, "Rtlfixer: Automatically fixing rtl syntax errors with large language models," 2024. [Online]. Available: https://arxiv.org/abs/2311.16543

[55] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, 2017.

[56] D. C. McElfresh, S. Khandagale, J. Valverde, V. P. C, G. Ramakrishnan, M. Goldblum, and C. White, "When do neural nets outperform boosted trees on tabular data?" in *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. [Online]. Available: https://openreview.net/forum?id=CjVdXey4zT

[57] OpenAI, "GPT-4," Mar. 2023. [Online]. Available: https://openai.com/research/gpt-4

[58] R. I. Bahar, A. K. Jones, S. Katkoori, P. H. Madden, D. Marculescu, and I. L. Markov, "Workshops on extreme scale design automation (esda) challenges and opportunities for 2025 and beyond," 2020. [Online]. Available: https://arxiv.org/abs/2005.01588

[59] X. Wang, Y. Jiang, N. Bach, T. Wang, Z. Huang, F. Huang, and K. Tu, "Automated concatenation of embeddings for structured prediction," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, C. Zong, F. Xia, W. Li, and R. Navigli, Eds. Online: Association for Computational Linguistics, Aug. 2021, pp. 2643–2660. [Online]. Available: https://aclanthology.org/2021.acl-long.206/

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: The central claims and motivation of the paper are articulated in the abstract and reiterated in the concluding paragraph of the introduction.

   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: The limitations of this work are addressed in the final paragraph of the conclusion section.

   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: [NA]

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: Comprehensive details of the experimental setup, including hyperparameters and dataset specifications, are provided in the results section. Additionally, the code has been made publicly available through an open-source release.

   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: [Yes]

Justification: The code required to reproduce the results has been open-sourced and is accessible at: https://github.com/ML4EDA/VeriLoC.git.

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: All training and testing details—including data splits, hyperparameters, their selection criteria, optimizer type, and related configurations—are thoroughly discussed in the 'Experimental Setting' subsection.

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: Due to cost considerations we don't do multiple bootstrap runs.

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: The necessary information regarding computational resources is provided in the Results section. This includes details on the type of compute infrastructure used, memory specifications, and execution time for the experiments, ensuring that the experiments can be reproduced reliably.

   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 research presented in the paper adheres to the NeurIPS Code of Ethics in all respects. It ensures transparency, reproducibility, and responsible use of data and models, with no ethical concerns related to data collection, experimental design, or potential downstream harm.

   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: [NA]

Justification: The work is for Quality of Results prediction for chip design

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: No such risks arise for this work.

    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: See Section 4.1. For the Synopsys Tool, we have a valid license to use.

    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: [Yes]

    Justification: See https://github.com/ML4EDA/VeriLoC.git

    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: [NA]

    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: [NA]

    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 significantly 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: We use a domain-adapted LLM fine-tuned on Verilog to generate embedding, see Section 3.

    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.

A IC Design

The design of modern ICs with billions of transistors and ever-more complex technology nodes is highly demanding. Thus, sophisticated tooling is essential in this domain known as electronic design automation (EDA). EDA tools are build upon strong notions of abstractions and hierarchical procedures. Next, we outline the industry-standard approach for these procedures. More details relevant for this work are also discussed in subsections in here.

System Specification, Architectural Design. Objectives and requirements for functionality, performance, and physical implementation are formulated. Modeling languages like SystemC can be used for a formalized approach.

Behavioral and Logic Design. The specification and architecture description are transformed into a behavioral model which describes inputs, outputs, timing behavior, etc. for the whole system. Toward that end, specific hardware description languages (HDLs) like Verilog are utilized. The abstraction level for this process is also often referred to as register-transfer level (RTL). To limit design time and efforts, third-party components, so-called IP modules, can be integrated at this stage.

Logic Synthesis. The behavioral model is transformed into a low-level circuit description, the gate-level netlist (GLN). This step requires a technology library for mapping from a generic circuit to the technology-specific circuit that is to be manufactured in the end.

Physical Design. The GLN is transformed into an actual physical layout of gates, memories, interconnects, etc. Given the high complexity of this stage, it is typically further divided into the following tasks: partitioning and/or floorplanning, power and ground delivery, placement, clock delivery, routing, and timing closure.

Verification and Signoff. The physical layout must be verified against various design and manufacturing rules, to ensure correct functionality and electrical behaviour. Once all rules are met, the design can be signed-off and taped-out, i.e., send out for fabrication, packaging, and testing.

Practical Challenge. Despite the general success of this compartmentalized approach, a key challenge remains: going through the full EDA stack end-to-end takes considerable time and efforts, in the range of months even for large teams. This challenge is know as the "productivity gap" and is expected to stay, especially for ever-more advanced technology nodes [58]. In an effort for best design quality, engineers often need to reiterate many times over key processes like placement and routing. Due to the very nature of the hierarchical tooling, the individual processes are often lacking detailed insights from prior stages and, more concerning, reasonable estimates for their impact on processes further down in the pipeline. In other words, there is a strong need to integrate well-informed quality assessment into early stages of the EDA pipeline—this reiterates the main motivation of this work at hand.

B Encoder-Decoder Architecture for Dimensionality Reduction

This model is designed to effectively compress high-dimensional embeddings into a compact latent space while preserving their semantic content. Below, we analyze its structure and key features:

B.1 Network Architecture

The encoder-decoder network consists of fully connected layers with batch normalization, dropout, and non-linear activation functions. The detailed architecture is shown in Table 6.

B.2 Regularization Techniques

To ensure robust learning and prevent overfitting, the following techniques are incorporated:

- **Dropout:** Applied with a 30% probability at multiple layers to prevent co-adaptation of neurons.
- **Batch Normalization:** Normalizes activations at each layer to stabilize training and improve convergence.
- **LeakyReLU:** Chosen over standard ReLU to avoid dead neurons and ensure a small gradient for negative values.

Table 6: Detailed architecture of the encoder-decoder network.

Layer Type	Number of Units	Activation Function	Additional Features
Encoder			
Fully Connected	4096	LeakyReLU ($\alpha = 0.01$)	BatchNorm, Dropout (30%)
Fully Connected	1024	LeakyReLU ($\alpha = 0.01$)	BatchNorm, Dropout (30%)
Fully Connected	Latent Dim (d)	LeakyReLU ($\alpha = 0.01$)	BatchNorm
Decoder			
Fully Connected	1024	LeakyReLU ($\alpha = 0.01$)	BatchNorm, Dropout (30%)
Fully Connected	4096	LeakyReLU ($\alpha = 0.01$)	BatchNorm, Dropout (30%)
Fully Connected	Input Dim	-	-

B.3 Optimization Strategy

The model is trained using the `AdamW` optimizer, which combines adaptive learning rates with weight decay regularization. Additional details include:

- **Learning Rate:** Set to a small value (10^{-4}) to ensure gradual convergence.
- **Reconstruction Loss:** The mean squared error (MSE) between the input embeddings and their reconstructed versions is minimized.
- **Weight Initialization:** All linear layers are initialized using `Xavier Uniform Initialization` to ensure balanced gradients at the start of training.

B.4 Training Procedure

The model is trained over 200 epochs using mini-batch gradient descent, with the following considerations:

- **Batch Size:** A batch size of 128 is chosen to balance memory efficiency and gradient stability.
- **Orthogonality Regularization:** Although not implemented in this iteration, orthogonality constraints on the encoder weights can further enhance disentanglement in the latent space.
- **Validation:** Validation loss on a separate test dataset is monitored to ensure the model generalizes to unseen data.

B.5 Usage in Downstream Tasks

The trained encoder produces latent embeddings that serve as input features for classification and regression tasks. These embeddings are compact, noise-robust, and retain essential semantic information from the original input.

B.6 Justification for Using the Last Hidden Layer in `VeriLoC`

Although `VeriLoC` focuses on line-level quality prediction, we first conducted a complementary ablation study on module-level embeddings to evaluate the predictive quality of hidden states extracted from different layers of the CL-Verilog model. As shown in Table 7, we observed that embeddings derived from the *last decoder layer* consistently achieved the highest accuracy across all quality-of-result (QoR) metrics—yielding the best R^2 scores and the lowest MAPE for both timing and congestion prediction. These results suggest that the final layer best captures high-level semantic and structural information necessary for downstream design-quality tasks.

Based on this empirical evidence, `VeriLoC` exclusively uses the last hidden layer to extract both line-level and module-level embeddings. This design choice ensures that the model benefits from the richest representation available, without introducing ambiguity or requiring additional architectural tuning to select among intermediate layers. Furthermore, the consistently superior performance of the final layer justifies avoiding ensemble or multi-layer fusion strategies, which may add unnecessary complexity without proportional gains.

Table 7: Prediction quality using embeddings derived from hidden states of the model, specifically examining the first (1st, 2nd, 3rd) and last (3rd-Last, 2nd-Last, Last) layers for *CL-Verilog*.

Hidden Layer	Timing		Congestion	
	R^2	MAPE	R^2	MAPE
1st	0.66	15.46	0.66	18.13
2nd	0.77	5.64	0.59	23.92
3rd	0.75	6.55	0.72	13.62
3rd-Last	0.85	5.88	0.53	22.34
2nd-Last	0.85	9.32	0.59	17.56
Last	0.89	5.57	0.74	11.66

Table 8: Impact of Hidden Dimension and Classifier on Line-Level Detection Performance for Congestion and Timing. Precision (P), Recall (R).

	Hidden Dim.	Congestion		Timing	
		P	R	P	R
XGB	32	0.76	0.64	0.77	0.71
	64	0.88	0.71	0.86	0.83
	128	0.94	0.78	0.94	0.94
	256	0.94	0.78	0.94	0.94
LGBM	32	0.77	0.64	0.79	0.71
	64	0.88	0.72	0.86	0.82
	128	0.94	0.79	0.96	0.94
	256	0.94	0.79	0.96	0.94

C Additional Ablation Studies

C.1 Role of Encoder and Choice of Hidden Dimensions

As noted in Section 3.3, we use an encoder-decoder architecture to reduce the dimensionality of raw line- and module-level embeddings before classification/regression. Without the encoder, we obtained F1-scores of less than 0.5. To determine the optimal hidden dimension for the encoder, we study different embedding dimensions across XGBoost and LightGBM. The results are summarized in Table 8.

We find that increasing the hidden dimension significantly improves both congestion and timing detection performance up to a dimension of 128. Beyond 128, however, there is no observable gain in performance, suggesting that further increasing the embedding size is not warranted and that 128 is the optimal hidden dimension, balancing performance and computational efficiency.

C.2 Impact of Comments in RTL

As opposed to intermediate representations like AIGs, RTL code also contains comments that we hypothesized to be useful in design quality predictions. Here, we evaluate the role of comments in `VeriLoC`'s accuracy. To this end, we conducted a study where we removed comments from both the train and test datasets and train `VeriLoC` classifiers on the resulting code. Table 9 compares line-level detection performance with (w) vs without (w/o) comments in the embeddings. The results show a modest but clear improvement in congestion and timing precision, recall, and F1-scores when comments are included. The benefits are starkest for FNN, where F1-scores increase from 0.71 to 0.77 for timing and 0.70 to 0.76 for congestion. XGBoost and LightGBM also benefit from comments, especially for congestion prediction with F1-scores increasing from 0.92 to 0.95 in the best case.

Table 9: Effect of Comments (C) in Module Embedding Generation on Line Detection Performance. Precision (P), Recall (R).

	Emb.	Congestion			Timing		
		P	R	F1	P	R	F1
FNN	w/o C	0.80	0.64	0.71	0.63	0.78	0.70
	w/ C	0.86	0.7	0.77	0.67	0.88	0.76
XGB	w/o C	0.93	0.76	0.84	0.91	0.92	0.91
	w/ C	0.94	0.78	0.85	0.94	0.94	0.94
LGBM	w/o C	0.95	0.77	0.85	0.93	0.91	0.92
	w/ C	0.94	0.79	0.86	0.96	0.94	0.95

Fig. 5 further supports these findings through saliency-based visualizations of batched vs individual line embeddings. Specifically, the saliency heatmaps reveal that, w/o comments, key structural elements like `always @(posedge clk)` receive a disproportionate amount of attention, whereas w/ comments, the model distributes attention more effectively across relevant components.

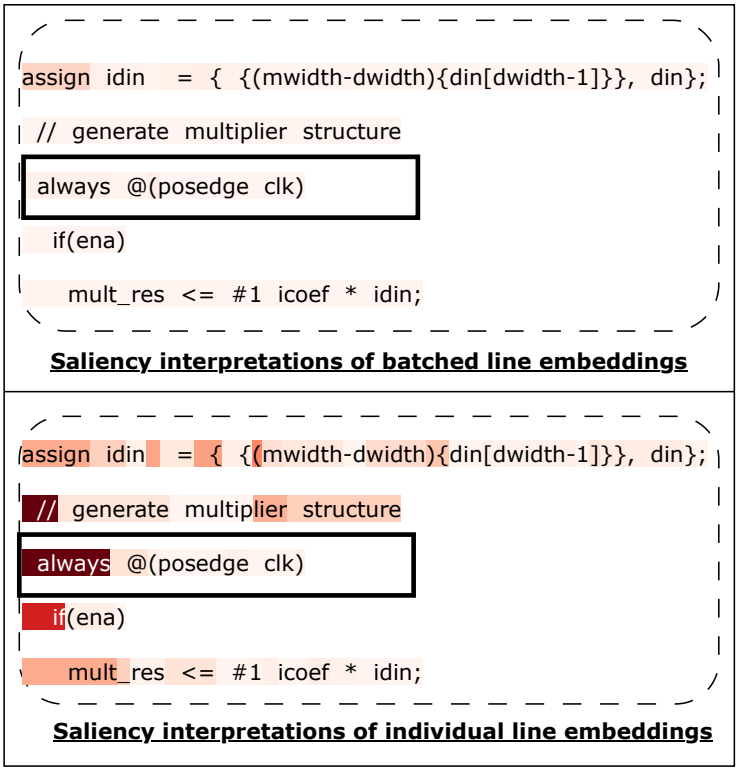

Figure 5: Saliency map comparing attention of `VeriLoC` model (bottom) vs. batched line embeddings (top).

C.3 Alternate Approaches

Recall that including neighboring lines of code can significantly improve performance, especially for congestion prediction. An alternate method for including local context, however, would be to obtain a single embedding for a batch of consecutive lines $B = \{l_{i-p}, \ldots, l_i, \ldots, l_{i+p}\}$ by passing

B through *CL-Verilog* as a single input. The model computes batched embeddings $z_{\text{batch}}(B)$, which natively capture inter-line dependencies via the LLM's attention mechanism.

Although appealing, the batched approach results in lower F1-scores, achieving at best an F1-score of 0.8 for congestion prediction; recall that `VeriLoC` achieves an F1-score of 0.86. Fig. 5 depicts the attention patterns of the model for congestion detection where the line `always @(posedge clk)` is critical. For the batched approach, the model's attention is dispersed across multiple unrelated tokens. Conversely, line-wise embeddings allow the model to prioritize the relevant tokens more effectively, as shown by the darker-red shades. This also aligns with the findings in [59], which emphasize the importance of modular embedding strategies for structured tasks.

D Case Study Using `GPT-4o` and `VeriLoC`

GPT-4o

Prompt: You are an expert in Verilog hardware design. Can you identify the congestion causing lines from the RTL module given below:

```
module aes_rcon(clk, kld, out);
input       clk;
input       kld;
output      [31:0]      out;
reg   [31:0]      out;
reg   [3:0] rcnt;
wire  [3:0] rcnt_next;

always @(posedge clk)
      if(kld)             out <= #1 32'h01_00_00_00;
      else          out <= #1 frcon(rcnt_next);

assign rcnt_next = rcnt + 4'h1;
always @(posedge clk)
      if(kld)             rcnt <= #1 4'h0;
      else          rcnt <= #1 rcnt_next;

function [31:0]  frcon;
input [3:0] i;
(* parallel_case *) case(i)
  4'h0: frcon=32'h01_00_00_00;
  4'h1: frcon=32'h02_00_00_00;
  4'h2: frcon=32'h04_00_00_00;
  4'h3: frcon=32'h08_00_00_00;
  4'h4: frcon=32'h10_00_00_00;
  4'h5: frcon=32'h20_00_00_00;
  4'h6: frcon=32'h40_00_00_00;
  4'h7: frcon=32'h80_00_00_00;
  4'h8: frcon=32'h1b_00_00_00;
  4'h9: frcon=32'h36_00_00_00;
  default: frcon=32'h00_00_00_00;
endcase
endfunction

endmodule
```

VeriLoC

```
module aes_rcon(clk, kld, out);
input       clk;
input       kld;
output      [31:0]      out;
reg   [31:0]      out;
reg   [3:0] rcnt;
wire  [3:0] rcnt_next;

always @(posedge clk)
      if(kld)             out <= #1 32'h01_00_00_00;
      else          out <= #1 frcon(rcnt_next);

assign rcnt_next = rcnt + 4'h1;
always @(posedge clk)
      if(kld)             rcnt <= #1 4'h0;
      else          rcnt <= #1 rcnt_next;

function [31:0]  frcon;
input [3:0] i;
(* parallel_case *) case(i)
  4'h0: frcon=32'h01_00_00_00;
  4'h1: frcon=32'h02_00_00_00;
  4'h2: frcon=32'h04_00_00_00;
  4'h3: frcon=32'h08_00_00_00;
  4'h4: frcon=32'h10_00_00_00;
  4'h5: frcon=32'h20_00_00_00;
  4'h6: frcon=32'h40_00_00_00;
  4'h7: frcon=32'h80_00_00_00;
  4'h8: frcon=32'h1b_00_00_00;
  4'h9: frcon=32'h36_00_00_00;
  default: frcon=32'h00_00_00_00;
endcase
endfunction

endmodule
```

Figure 6: The lines of code highlighted by GPT-4o and `VeriLoC` for congestion metrics on `aes_rcon` design.

As a part of the case study, to understand the capability of generic LLMs to analyze RTL code, we took `GPT-4o` [57] and prompted it to report the line numbers in the code responsible for timing and congestion, respectively. We also showcase `VeriLoC` performance for the same codes. `GPT-4o` was unable to detect the correct line numbers, i.e, showed false positives, whereas `VeriLoC` has shown significant results when compared to the ground truth obtained using the EDA tool.

Fig. 6–9 show such examples with highlighted text being the line number reported by `GPT-4o` and `VeriLoC`, respectively. Fig. 6 refers to the `aes_rcon` design, where `GPT-4o` predicts every line as critical for congestion issues, whereas `VeriLoC` selectively highlights the correct lines related to congestion. Fig. 7 again shows an inability of `GPT-4o`, now for the task of timing prediction.

GPT-4o

Prompt: You are an expert in Verilog hardware design. Can you identify the timing issue causing lines from the RTL module given below:

```
module ibex_csr (
    clk_i,
    rst_ni,
    wr_data_i,
    wr_en_i,
    rd_data_o,
    rd_error_o
);
    parameter [31:0] Width = 32;
    parameter [0:0] ShadowCopy = 1'b0;
    parameter [Width - 1:0] ResetValue = 1'sb0;
    input wire clk_i;
    input wire rst_ni;
    input wire [Width - 1:0] wr_data_i;
    input wire wr_en_i;
    output wire [Width - 1:0] rd_data_o;
    output wire rd_error_o;
    reg [Width - 1:0] rdata_q;
    always @(posedge clk_i or negedge rst_ni)
        if (!rst_ni)
            rdata_q <= ResetValue;
        else if (wr_en_i)
            rdata_q <= wr_data_i;
    assign rd_data_o = rdata_q;
    generate
        if (ShadowCopy) begin : gen_shadow
            reg [Width - 1:0] shadow_q;
            always @(posedge clk_i or negedge rst_ni)
                if (!rst_ni)
                    shadow_q <= ~ResetValue;
                else if (wr_en_i)
                    shadow_q <= ~wr_data_i;
            assign rd_error_o = rdata_q != ~shadow_q;
        end
        else begin : gen_no_shadow
            assign rd_error_o = 1'b0;
        end
    endgenerate
endmodule
```

VeriLoC

```
module ibex_csr (
    clk_i,
    rst_ni,
    wr_data_i,
    wr_en_i,
    rd_data_o,
    rd_error_o
);
    parameter [31:0] Width = 32;
    parameter [0:0] ShadowCopy = 1'b0;
    parameter [Width - 1:0] ResetValue = 1'sb0;
    input wire clk_i;
    input wire rst_ni;
    input wire [Width - 1:0] wr_data_i;
    input wire wr_en_i;
    output wire [Width - 1:0] rd_data_o;
    output wire rd_error_o;
    reg [Width - 1:0] rdata_q;
    always @(posedge clk_i or negedge rst_ni)
        if (!rst_ni)
            rdata_q <= ResetValue;
        else if (wr_en_i)
            rdata_q <= wr_data_i;
    assign rd_data_o = rdata_q;
    generate
        if (ShadowCopy) begin : gen_shadow
            reg [Width - 1:0] shadow_q;
            always @(posedge clk_i or negedge rst_ni)
                if (!rst_ni)
                    shadow_q <= ~ResetValue;
                else if (wr_en_i)
                    shadow_q <= ~wr_data_i;
            assign rd_error_o = rdata_q != ~shadow_q;
        end
        else begin : gen_no_shadow
            assign rd_error_o = 1'b0;
        end
    endgenerate
endmodule
```

Figure 7: The lines of code highlighted by GPT-4o and `VeriLoC` for timing metrics on `ibex_csr` design.

Although it has highlighted the lines related to timing impact, namely the first `always` block, it also reports the next blocks as potential candidates. In Fig. 8, `GPT-4o` picks `registers` definitions as a potential reason for congestion and skips the actual criticial line containing the additions operation. Finally, in Fig. 9 `GPT-4o` was unable to pick the correct `always` block responsible for timing issues.

GPT-4o

Prompt: You are an expert in Verilog hardware design. Can you identify the congestion causing lines from the RTL module given below:

```
wire [mwidth-1:0] idin;
wire [mwidth-1:0] icoef;

reg  [mwidth -1:0] mult_res;
wire [rwidth -1:0] ext_mult_res;

//
// module body
//
assign icoef = { {(mwidth-cwidth){coef[cwidth-1]}}, coef};
assign idin  = { {(mwidth-dwidth){din[dwidth-1]}}, din};

// generate multiplier structure
always @(posedge clk)
   if(ena)
       mult_res <= #1 icoef * idin;

assign ext_mult_res = { {3{mult_res[mwidth-1]}}, mult_res};

// generate adder structure
always @(posedge clk)
  if(ena)
      if(dclr)
          result <= #1 ext_mult_res;
      else
          result <= #1 ext_mult_res + result;
```

VeriLoC

```
wire [mwidth-1:0] idin;
wire [mwidth-1:0] icoef;

reg  [mwidth -1:0] mult_res;
wire [rwidth -1:0] ext_mult_res;

//
// module body
//
assign icoef = { {(mwidth-cwidth){coef[cwidth-1]}}, coef};
assign idin  = { {(mwidth-dwidth){din[dwidth-1]}}, din};

// generate multiplier structure
always @(posedge clk)
   if(ena)
       mult_res <= #1 icoef * idin;

assign ext_mult_res = { {3{mult_res[mwidth-1]}}, mult_res};

// generate adder structure
always @(posedge clk)
  if(ena)
      if(dclr)
          result <= #1 ext_mult_res;
      else
          result <= #1 ext_mult_res + result;
```

Figure 8: The lines of code highlighted by GPT-4o and `VeriLoC` for congestion metrics on `dct_mac` design.

GPT-4o

Prompt: You are an expert in Verilog hardware design. Can you identify the timing issue causing lines from the RTL module given below:

```
      output [11:0] dout;
      output        douten; // data-out enable

      //
      // variables
      //

      reg ld_zigzag;
      reg [11:0] sresult [63:0]; // store results for zig-zagging
      //
      // module body
      //

      always @(posedge clk)
        if(ena)
          ld_zigzag <= #1 dstrb;

      assign douten = ld_zigzag;

      integer n;

      always @(posedge clk)
        if(ena)
          if(ld_zigzag)   // reload results-register file
          begin
             sresult[63] <= #1 din_00;
             sresult[62] <= #1 din_01;
                         .
                         .
                         .
                         .
                         .
                         .
             sresult[09] <= #1 din_47;
             sresult[08] <= #1 din_56;
             sresult[03] <= #1 din_57;
             sresult[02] <= #1 din_67;
             sresult[01] <= #1 din_76;
          end
        else       // shift results out
          for (n=1; n<=63; n=n+1) // do not change sresult[0]
            sresult[n] <= #1 sresult[n -1];

      assign dout = sresult[63];
endmodule
```

VeriLoC

```
      //
      // module body
      //

      always @(posedge clk)
        if(ena)
          ld_zigzag <= #1 dstrb;

      assign douten = ld_zigzag;
      integer n;

      always @(posedge clk)
        if(ena)
          if(ld_zigzag)   // reload results-register file
          begin
             sresult[63] <= #1 din_00;
                         .
                         .
                         .
             sresult[45] <= #1 din_32;
             sresult[44] <= #1 din_41;
             sresult[43] <= #1 din_50;
                         .
                         .
                         .
             sresult[33] <= #1 din_25;
             sresult[32] <= #1 din_34;
             sresult[31] <= #1 din_43;
             sresult[30] <= #1 din_52;
             sresult[27] <= #1 din_71;
             sresult[26] <= #1 din_62;
             sresult[19] <= #1 din_36;
             sresult[18] <= #1 din_45;
             sresult[17] <= #1 din_54;
             sresult[16] <= #1 din_63;
             sresult[15] <= #1 din_72;
             sresult[14] <= #1 din_73;
             sresult[13] <= #1 din_64;
             sresult[12] <= #1 din_55;
             sresult[11] <= #1 din_46;
             sresult[10] <= #1 din_37;
             sresult[09] <= #1 din_47;
             sresult[08] <= #1 din_56;
             sresult[03] <= #1 din_57;
             sresult[02] <= #1 din_67;
             sresult[01] <= #1 din_76;
          end
        else       // shift results out
          for (n=1; n<=63; n=n+1) // do not change sresult[0]
            sresult[n] <= #1 sresult[n -1];

      assign dout = sresult[63];
endmodule
```

Figure 9: The lines of code highlighted by GPT-4o and `VeriLoC` for timing metrics on `zigzag` design.