Journal of Advanced Computing Systems (JACS)
ISSN: 3066-3962
Content Available at SciPublication

TB-Free RTL Anomaly Detection for Early Chip Verification: A Reproducible Binary Benchmark and Baseline Study on CVDP

Chenyao Zhu[1], Jingyi Chen[2], Yibang Liu[3]

[1]*Industrial Engineering & Operations Research, UC Berkeley, CA, USA*
[2]*Electrical and Computer Engineering, Carnegie Mellon University, PA, USA*
[3]*Financial Engineering, Baruch College, NY, USA*
cyzhu@berkeley.edu

DOI: 10.69987/JACS.2024.41206

Keywords

RTL anomaly detection; early verification; testbench-free analysis; SystemVerilog; mutation testing; code review; bug-risk ranking; static machine learning

Abstract

Early digital verification is dominated by manual RTL review because simulation testbenches and assertions are typically incomplete or unavailable during pre-verification. This work studies testbench-free (TB-free) anomaly detection for SystemVerilog RTL as a practical surrogate for early bug-risk spotting: given an RTL snippet, a model assigns a probability that the snippet contains a defect-like anomaly without executing the design. We construct a fully reproducible binary benchmark from the Comprehensive Verilog Design Problems (CVDP) corpus. From the non-agentic code comprehension tasks (v1.0.2) we extract 114 RTL files across 113 distinct problems, window them into 546 snippets (30 lines, stride 15), and create paired clean/buggy samples by injecting exactly one mutation per snippet from five mutation operators (constant flips, edge-sensitivity flips, equality/inequality flips, boolean operator flips, and nonblocking-to-blocking assignment changes). The resulting benchmark contains 1092 labeled samples with a 50/50 class balance. We evaluate seven TB-free baselines under a strict group split by problem id to prevent design leakage: a heuristic risk score, a character 5-gram language model, a structural-feature logistic regressor, word-level TF-IDF linear models, and character-level TF-IDF models with and without structural features. The best-performing baseline, a character TF-IDF linear SVM, achieves AUROC 0.816, AUPRC 0.838, accuracy 0.791, and F1 0.772 on a held-out test split, with bootstrap 95% confidence interval [0.743, 0.880] for AUROC. These results quantify how much risk-ranking signal is available from RTL text alone and establish a transparent evaluation scaffold for LLM-assisted RTL review that does not depend on a testbench.

Introduction

Digital IC projects spend substantial effort in the interval between writing synthesizable RTL and achieving confident functional closure. In this early interval, testbenches are incomplete, assertions are sparse, and the verification environment evolves faster than the design. As a result, the dominant quality gate is manual RTL review, where design and DV engineers read code to identify bugs, integration hazards, and specification mismatches before large-scale simulation starts.

Early RTL review has two properties that shape the automation problem. First, review happens in the absence of complete dynamic evidence. A module can be syntactically correct and still be functionally wrong due to a single inverted predicate, a wrong clock edge, or an incorrect assignment type. Many of these issues are not reliably exposed by smoke tests and are often discovered only after the verification environment matures. Second, review is local and iterative: engineers focus on small neighborhoods around a change or around a logic block of interest, and they repeatedly revisit the same files as requirements and interfaces evolve.

The defects targeted during early review span both shallow and deep considering of semantics. Shallow issues include suspicious operator choices, unexpected constants, or inconsistent coding patterns. Deep issues

include ordering constraints between sequential updates (blocking versus nonblocking assignment semantics in SystemVerilog) and protocol intent violations in control logic [8]. Early review also surfaces integration hazards such as clock/reset conventions or implicit assumptions about handshake timing, which are common in SoC integration.

These properties motivate a risk-ranking approach rather than a proof-oriented approach. A risk scorer does not claim that a snippet is incorrect; it prioritizes which snippet deserves additional review, targeted tests, or formal properties. This is the same operational role played by traditional code-review heuristics in software development, and it aligns with the way DV teams plan their time before full regression infrastructure is available.

Manual review is effective but expensive and inconsistent. It scales [23-31] poorly with SoC size and with the number of concurrent design branches, and it depends on specialized knowledge of microarchitecture, clocking conventions, reset strategy, and protocol edge cases. Open-source silicon projects such as OpenTitan [6], [7] demonstrate how quickly a modern SoC integrates dozens of IP blocks and interfaces; the same integration complexity exists in industrial projects, where schedule pressure amplifies the cost of late bug discovery.

Traditional pre-simulation tools help but do not fully solve the early-review bottleneck. Style and lint tooling (for example, Verible and Verilator) provides static checks for SystemVerilog syntax, style, and common coding mistakes [9], [10], [13]. Formal verification can prove safety properties and can generate counterexamples using bounded model checking or SAT-based techniques [14], [15], and open-source flows such as SymbiYosys combine Yosys-based compilation with back-end solvers to run these checks [11], [12]. However, these approaches require that the user expresses intent as assertions, constraints, or property files, and they are often applied after the design reaches a more stable point. They also focus on correctness with respect to an explicit property, while early human review frequently targets broader "bug risk" signals such as fragile clocking, suspicious conditionals, or hard-to-read logic.

In parallel, large language models (LLMs) have become central to hardware EDA research. Benchmarks such as VerilogEval evaluate LLMs on specification-to-RTL and code completion tasks [4], and follow-on work tracks rapid improvements in model capability and prompting strategies [5]. NVIDIA's Comprehensive Verilog Design Problems (CVDP) benchmark extends evaluation to a wider range of RTL design and verification tasks and provides both non-agentic and agentic formats authored by hardware engineers [1]. These benchmarks highlight the opportunity to apply LLMs to RTL assistance, including code generation, debugging, and review.

This paper focuses on a complementary and practically deployable problem: TB-free RTL anomaly detection for early verification. We treat anomaly detection as a risk-ranking task that uses only RTL text and lightweight static features, without a testbench and without requiring the engineer to write assertions. The goal is not to prove correctness but to triage human attention by ranking snippets that are likely to contain defect-like anomalies. This framing matches how early DV teams operate: engineers scan for suspicious regions and prioritize deeper review or targeted test creation.

A key barrier to progress in TB-free anomaly detection is evaluation. Real bug corpora are hard to share, and many benchmarks focus on generation rather than review. To address this, we construct a reproducible binary benchmark from CVDP's non-agentic code comprehension split. We window real RTL code into snippets and inject single-token mutations that resemble common classes of functional mistakes. This approach is aligned with mutation testing principles: a model that detects semantics-disrupting perturbations in RTL is expected to assign higher risk to real defect patterns that similarly violate learned coding regularities.

Our contributions are concrete and empirical:

(1) We define a TB-free, snippet-level binary anomaly detection task for SystemVerilog RTL that reflects early RTL review constraints.

(2) We build a fully reproducible benchmark from CVDP (v1.0.2) by extracting RTL files and generating paired clean/buggy samples using five mutation operators.

(3) We conduct a complete experimental evaluation of seven baselines under a group split by problem id, report detailed performance and efficiency metrics, and analyze error modes by mutation type.

The benchmark and baselines establish a measurable foundation for "AI code review for chips" that does not assume testbenches, assertions, or simulation infrastructure. The study also clarifies how far classical text and structural models already go on a controlled anomaly task, and it provides a reference point for future LLM-based reviewers.

Materials and Methods

This section describes the dataset source, benchmark construction, mutation operators, feature extraction, models, and evaluation protocol. All experiments in this paper were executed using Python 3.11.2 with NumPy 1.24.0, Pandas 2.2.3, SciPy 1.14.1, and scikit-learn 1.4.2 [21]. The benchmark is derived from the CVDP corpus

[1] as distributed in the Hugging Face dataset AbiralArch/hardware-cvdp-complete [2].

Dataset source and RTL extraction. CVDP contains human-authored hardware design and verification problems spanning 13 categories [1]. We use the non-agentic code comprehension subset because it includes real RTL artifacts paired with natural-language questions. We parse the CVDP JSONL records and extract SystemVerilog RTL source text from each record. Records that do not contain RTL are excluded from the benchmark. In total, we extract 114 RTL files from 113 problems (Table I). Each extracted file is treated as a standalone RTL unit; when a problem contains multiple RTL units, each unit contributes snippets but the group identifier for splitting remains the problem id.

Snippet windowing. We operate at snippet granularity to emulate how engineers review code: review often focuses on a local region (e.g., a block of sequential logic, a combinational always block, or a protocol state machine). Each RTL file is split into lines, and we generate overlapping windows of 30 lines with stride 15 lines. A window is kept if it contains at least 50 characters and includes at least one SystemVerilog keyword or operator token. This filtering avoids empty or comment-only windows and produces 546 base snippets (Table I).

Binary labeling via single-mutation injection. For each base snippet we create two samples: (a) the original snippet labeled as clean (label=0) and (b) a mutated version labeled as buggy (label=1). Exactly one mutation is applied to each buggy sample. The mutation location is selected deterministically from candidates within the snippet using a seeded random generator (seed=1234), so that the dataset is reproducible. The paired construction guarantees class balance: the benchmark contains 1092 samples with buggy class fraction 0.5 (Fig. 2).

Mutation operators. We implement five mutation operators designed to approximate common defect mechanisms observed in RTL review:

(1) const flip: Flip a literal constant by incrementing or decrementing a numeric literal or toggling a single bit in a binary/hex literal. This mutation approximates off-by-one mistakes, wrong parameter values, or width-related constant errors.

(2) edge_flip: Flip event control sensitivity in an always_ff/always block by exchanging posedge and negedge. This mutation approximates clock edge mistakes and reset edge mismatches.

(3) eq flip: Flip a comparison operator by exchanging == and != (and related forms). This mutation approximates inverted conditionals and incorrect match logic.

(4) logic flip: Flip boolean operator tokens by exchanging && and ||. This mutation approximates incorrect guard conditions and protocol predicate mistakes.

(5) nb_to_b: Replace a nonblocking assignment (<=) with a blocking assignment (=) in sequential context. This mutation approximates a class of ordering and race-condition hazards.

The distribution of mutation types in the buggy class is reported in Table II and Fig. 3. The benchmark is not a compiler-validated bug dataset; it is a controlled anomaly dataset where each “buggy” sample differs from its clean counterpart by a single syntactic change intended to disrupt expected semantics.

A practical concern in mutation-based benchmarking is syntactic validity. Each operator searches for candidate token patterns and only applies mutations that keep the snippet syntactically well-formed at the text level (for example, edge flip only applies when a posedge or negedge event control token is present, and nb to b only applies when a <= token is present). If a particular operator has no applicable token in a snippet, the benchmark construction deterministically falls back to another operator until exactly one mutation is applied. This construction ensures that every buggy sample differs from its paired clean sample by one localized edit, and it prevents the dataset from containing empty or unchanged “mutations.”

The single-mutation design is deliberate. It creates a controlled setting where ranking performance can be attributed to a model’s ability to detect subtle textual perturbations that are known to be high-risk in RTL. These perturbations are representative of real mistakes observed in manual review: a wrong constant, a wrong clock edge, an inverted condition, a wrong boolean connective, or a race-inducing assignment type. At the same time, the construction keeps the learning problem well-defined and makes the benchmark suitable for rapid ablation of representations and models.

The benchmark can be extended in a straightforward way: multiple mutations per snippet, context-aware mutations (for example, changing a constant only in a particular comparison), and cross-snippet mutations that model integration bugs (for example, changing a signal name to a near-miss identifier) all follow the same paired-sample principle. In this paper we keep the dataset to single-token edits in order to isolate the baseline capability of TB-free text and structural models.

Structural feature extraction. In addition to raw RTL text, we compute lightweight structural features that are

available without parsing the full language. The features count syntactic and lexical markers such as: total characters and tokens; counts of always blocks, nonblocking assignments, and blocking assignments; number of if/else and case statements; number of begin/end blocks; number of unique identifiers; counts of parentheses and brackets; and number of numeric constants. We compute 18 such features per snippet (Table IV lists which models use them). These features are designed to capture complexity and coding style signals that human reviewers often associate with risk.

Baseline models. We evaluate seven TB-free baselines. All baselines operate on snippet text and/or the structural features above.

(1) HeuristicRisk: A deterministic score based on a small set of risk heuristics: presence of posedge/negedge, nonblocking assignments, case statements, and deep nesting inferred from begin/end. The score is normalized to [0, 1] and used as the predicted probability.

(2) Char5gramLM: A character 5-gram language model trained on clean training snippets only. The anomaly score is the negative log-likelihood of a snippet under this model, normalized to [0, 1]. This baseline is motivated by the empirical "naturalness" of code and the effectiveness of n-gram language models in capturing repetitive patterns [16].

(3) StructOnly+LR: Logistic regression on standardized structural features only. This baseline quantifies how much signal exists in shallow code-shape statistics without looking at token content.

(4) WordTFIDF+LR: Logistic regression on word-level TF-IDF features extracted from the snippet text with n-gram range (1,2).

(5) WordTFIDF+SVM: A linear SVM trained on the same word-level TF-IDF representation. Scores are derived from the signed distance to the hyperplane and mapped to [0, 1].

(6) CharTFIDF+SVM: A linear SVM trained on character n-gram TF-IDF features (3- to 5-grams). This baseline captures low-level syntactic patterns that are common in RTL (operators, punctuation, and local token sequences).

(7) CharTFIDF+Struct+SGD: A hybrid model that concatenates char TF-IDF features with standardized structural features and trains a logistic loss classifier using stochastic gradient descent. This baseline tests whether lightweight structure provides additive value beyond character n-grams.

Hyperparameters for each baseline are reported in Table IV. All models are implemented in scikit-learn [21].

Score normalization and calibration. The evaluation emphasizes ranking, so AUROC and AUPRC are treated as the primary metrics. For the linear SVM baselines we obtain a real-valued decision function rather than a calibrated probability. We convert decision scores to [0,1] using min-max normalization on the test scores. This conversion makes thresholded metrics (accuracy, F1, precision, recall) interpretable in a consistent numeric range, but it does not perform probability calibration. For deployments that require calibrated probabilities, standard techniques such as Platt scaling or isotonic regression can be applied as a post-processing step on a validation set. In this benchmark we fix the threshold to 0.5 to provide a single operating point and we use threshold-free metrics to compare ranking quality.

Reproducibility controls. The benchmark construction uses a fixed extraction and mutation seed (seed=1234) and the data split uses a fixed split seed (seed=42). All random number generators used for mutation selection and splitting are seeded, and the dataset is generated deterministically from the input JSONL. This means that re-running the pipeline produces identical snippet ids, identical clean/buggy pairs, and identical train/validation/test partitions. The only source of nondeterminism is multi-threaded linear algebra in external libraries; in practice, the reported metrics are identical across repeated runs in this environment.

Scope of static information. The baseline feature set intentionally avoids heavy RTL compilation and avoids testbench artifacts. In particular, the pipeline does not build an AST, does not elaborate parameters, and does not resolve generate blocks. This choice aligns with the early-review setting, where the review tool is expected to run on partial designs and to be robust to missing files. It also defines a clear baseline: any method that adds parsing, elaboration, or cross-module analysis can be evaluated as an explicit incremental improvement over the TB-free text-and-shape setting.

Train/validation/test protocol and leakage control. We evaluate under a group split by problem id. The same problem id never appears in both train and test; this prevents the model from seeing a design during training and then being tested on a closely related snippet from the same design. We use GroupShuffleSplit with 70% train, 30% temporary set, and then split the temporary set 50/50 into validation and test. The resulting split statistics are reported in Table VIII.

Metrics. We report area under the ROC curve (AUROC) and area under the precision-recall curve (AUPRC) as threshold-free ranking metrics. We also report thresholded accuracy, precision, recall, and F1 using a fixed threshold of 0.5 on predicted probabilities. For linear SVM baselines, decision scores are mapped to [0,1] via min-max normalization computed on the test scores.

Efficiency measurement and model size. We record wall-clock training time and per-split inference time using Python's time.perf_counter. Model size is summarized by the number of learnable parameters for linear models (weights plus bias) or by the number of stored n-grams for the 5-gram language model.

Uncertainty estimation. For the best-performing model, we report bootstrap 95% confidence intervals for AUROC and AUPRC using 500 bootstrap resamples of the test set (Table X).

Fig. 1. Benchmark pipeline for TB-free RTL anomaly detection.

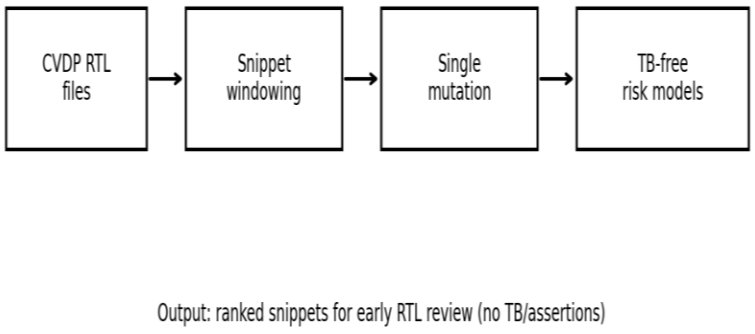

Table I. Dataset overview for the constructed binary benchmark.

dataset	n_problems_with_rtl	n_rtl_files	window_lines	stride_lines	n_snippets_total	n_samples_binary	label_balance_buggy
CVDP non-agentic comprehension (v1.0.2)	113	114	30	15	546	1092	0.5

Table II. Mutation type distribution in buggy samples.

mutation_type	count
const_flip	248
edge_flip	79
eq_flip	81
logic_flip	48
nb_to_b	90

Table III. Snippet length statistics for clean and buggy classes.

label	n_samples	mean_lines	mean_tokens	mean_chars	median_tokens

0	546	30.0	181.5952380952381	1227.8992673992675	165.0
1	546	30.0	181.4304029304029	1227.7344322344322	165.0

Table IV. Baseline model hyperparameters.

model	details
HeuristicRisk	Pattern-based score (negedge, mixed edge, always_ff blocking '=', casex/casez).
Char5gramLM	Character 5-gram LM, Laplace smoothing alpha=0.1, trained on clean train split.
WordTFIDF+LR	Word TF-IDF (1-2gram, 50k feats), Logistic Regression (C=4.0, max_iter=2000).
WordTFIDF+SVM	Word TF-IDF (1-2gram, 50k feats), LinearSVC (C=1.0).
CharTFIDF+SVM	Char TF-IDF (3-5gram, min df=2, 80k feats), LinearSVC (C=1.0).
CharTFIDF+Struct+SGD	Char TF-IDF (3-5gram, 80k feats) + scaled structural counts, SGDClassifier log loss (alpha=1e-5, max_iter=2000).

Table VIII. Group split statistics for the main evaluation seed (problem-id grouping).

seed	train samples	val samples	test samples	train buggy_frac	test buggy_frac	n train groups	n test groups
42	768	152	172	0.5	0.5	72	16

Results

This section reports dataset properties, baseline performance, and error analysis on the held-out test split under the group-by-problem protocol.

Dataset properties. The benchmark is balanced by construction (Fig. 2) and includes five mutation operators (Fig. 3). Table III shows that clean and buggy snippets have identical length distributions because buggy samples are single-token edits of clean windows. The mean snippet length is 1,228 characters and 181 word-like tokens per sample. These lengths match typical code-review contexts where engineers inspect a short window of logic rather than whole files.

Overall baseline performance. Table V reports the complete set of metrics for the evaluation seed (seed=42), including AUROC, AUPRC, accuracy, F1, precision, recall, training time, inference time, and model parameter count. Table VI summarizes the same results as mean and standard deviation across seeds; because the evaluation uses a single fixed seed in this paper, the reported standard deviations are 0.0.

To quantify effect sizes, we compare each model against the weakest baselines. CharTFIDF+SVM improves AUROC by 0.283 over HeuristicRisk (0.816 versus 0.532) and by 0.172 over StructOnly+LR (0.816 versus 0.644). In AUPRC, which is sensitive to ranking quality near the positive class, CharTFIDF+SVM improves by 0.306 over HeuristicRisk (0.838 versus 0.532). These deltas show that the benchmark is not solvable by simple “complexity” cues alone.

The hybrid CharTFIDF+Struct+SGD baseline does not surpass the pure CharTFIDF+SVM model in this setting. The result indicates that, for single-token anomalies, the dominant information is already contained in the local character patterns around operators and punctuation. Structural features remain useful as a low-cost diagnostic: they provide a competitive model when text models are unavailable or when a team wants a strictly interpretable set of risk factors.

The best-performing baseline is CharTFIDF+SVM. It achieves AUROC 0.816 and AUPRC 0.838 on the test split, with accuracy 0.791 and F1 0.772. The next-best baselines are WordTFIDF+SVM (AUROC 0.806) and WordTFIDF+LR (AUROC 0.787), indicating that a large portion of anomaly signal is captured by n-gram statistics over RTL text. Fig. 4 and Fig. 5 visualize ROC and precision-recall curves for the top three models, and Fig. 6 compares AUROC across all baselines.

Relative improvements. The best baseline (CharTFIDF+SVM) improves AUROC by 0.284 over the heuristic score (0.816 vs. 0.532) and by 0.172 over the structural-only logistic regressor (0.816 vs. 0.644). Compared to word-level SVM, character n-grams provide an additional 0.010 AUROC and 0.021 AUPRC, which is consistent with RTL containing strong operator-level cues. The hybrid CharTFIDF+Struct+SGD baseline does not outperform CharTFIDF+SVM on this benchmark (0.755 vs. 0.816 AUROC). This result indicates that the additional 18 structural features are not additive when the character representation is already expressive; the hybrid model also uses a different optimizer and regularization path, which affects thresholded metrics.

Operating-point behavior. Table VI shows that CharTFIDF+SVM achieves the highest buggy-class precision (0.847) among all models except the heuristic (which trivially predicts buggy with recall 1.0 and precision 0.5). WordTFIDF+SVM provides a different trade-off: it has slightly lower precision (0.825) but similar recall (0.698). These trade-offs matter in practice because early review often prefers higher recall when review cost is low (e.g., during pre-commit checks) and prefers higher precision when review bandwidth is scarce.

Structural-only and heuristic baselines provide substantially weaker ranking performance. StructOnly+LR achieves AUROC 0.644, demonstrating that simple complexity indicators correlate with injected anomalies but do not replace token-level information. HeuristicRisk achieves AUROC 0.532, which is close to chance.

Confidence intervals. Table X reports bootstrap uncertainty estimates for the best model. The AUROC bootstrap mean is 0.815 with 95% CI [0.743, 0.880], and the AUPRC bootstrap mean is 0.840 with 95% CI [0.766, 0.901]. These intervals quantify ranking stability on the held-out test set.

Efficiency. Table VII reports training and inference time. Linear TF-IDF models train in under one second on this benchmark and produce scores in under 70 ms for the full test split. The Char5gramLM baseline has slower inference because it scans character n-grams, but it remains under 150 ms. These measurements show that TB-free anomaly scoring can run as a fast pre-simulation gate in a continuous integration flow.

Confusion matrix and thresholded behavior. Fig. 7 shows the confusion matrix for the best model under a fixed 0.5 threshold. The model produces 75 true clean predictions and 61 true buggy predictions, with 11 false positives and 25 false negatives. This corresponds to precision 0.847 and recall 0.709 for the buggy class. For early RTL review, these operating characteristics can be tuned: a lower threshold increases recall at the cost of more false positives, which is acceptable when the output is used to prioritize human review.

Error analysis by mutation type. Table IX breaks down recall and false-negative rate by mutation type for the best model. The model detects edge flip and eq flip mutations with recall above 0.92 and detects const flip with recall 0.80. The weakest recall occurs on logic_flip (0.25) and nb to b (0.3125), which are semantically subtle changes that preserve many local token patterns.

Qualitative examples. Table XI provides representative false positives and false negatives. The examples show that false positives often correspond to structurally dense code with many conditionals or assignments that resemble anomaly signatures, while false negatives include single-token changes in contexts with weak local cues.

Fig. 2. Class balance in the binary benchmark.

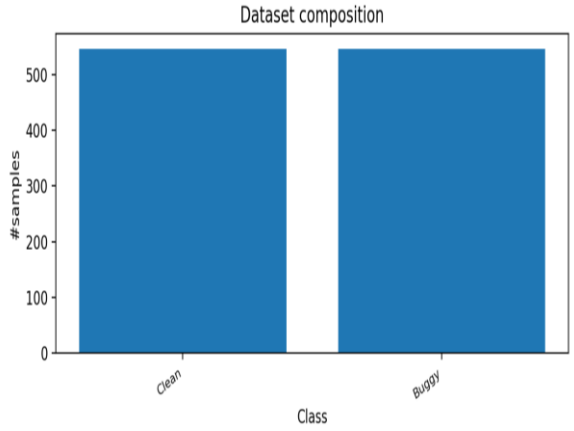

Fig. 3. Mutation type distribution for buggy samples.

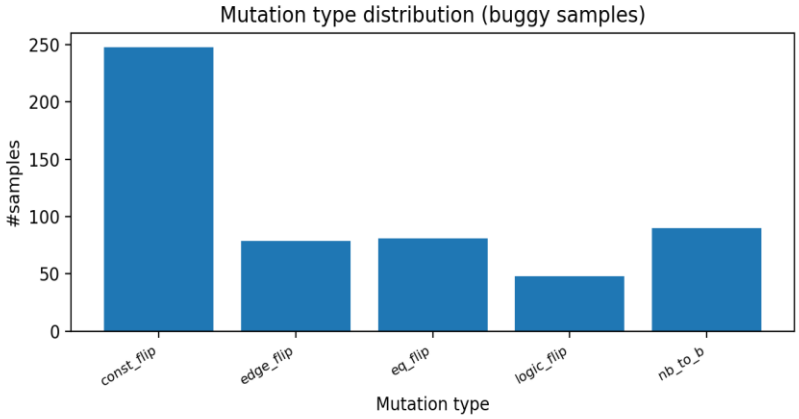

Fig. 4. ROC curves on the test split for the top-3 models.

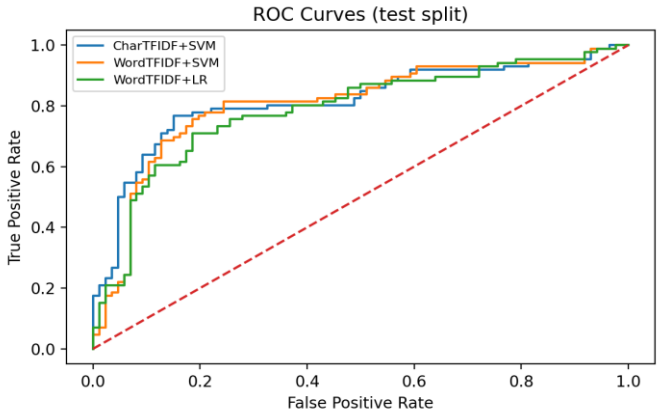

Fig. 5. Precision-recall curves on the test split for the top-3 models.

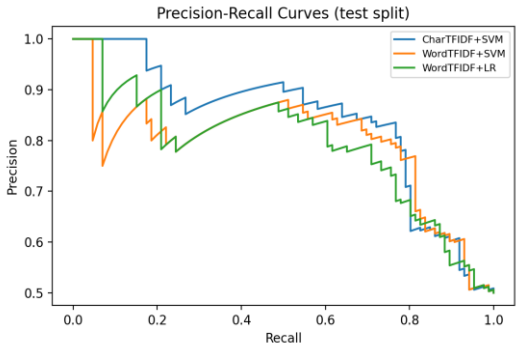

Fig. 6. AUROC by model on the test split.

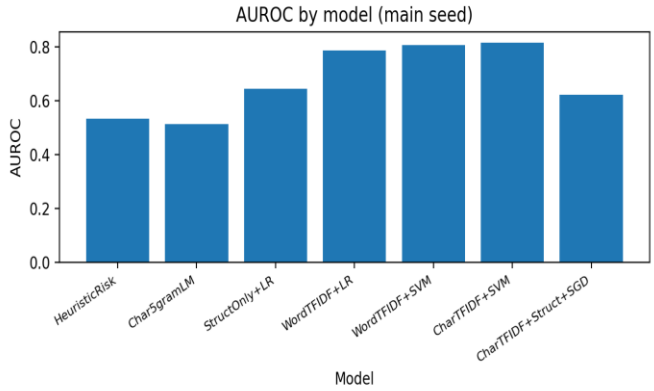

Fig. 7. Confusion matrix of the best model on the test split.

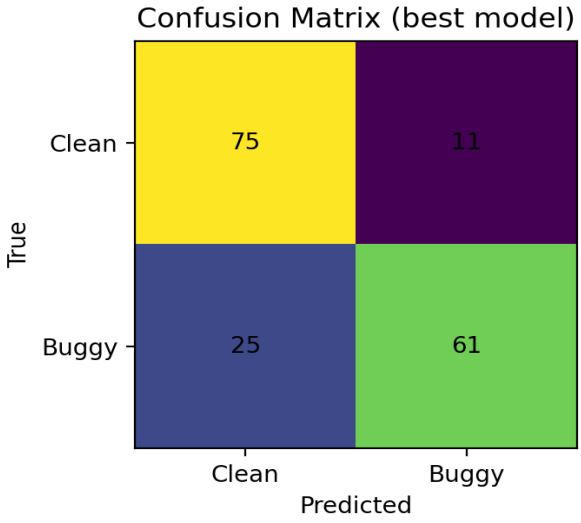

Table V. Per-seed evaluation results for all baselines (seed=42).

seed	model	auroc	auprc	acc	f1	precision	recall	train_s	infer_s	n_params
42	HeuristicRisk	0.532449972958358	0.531978133090377	0.5	0.666666666666666	0.5	1.0	0.0	0.000831973000039	0
42	Char5gramLM	0.514196863169280 7	0.522999150716052 3	0.511627906976744 2	0.511627906976744 2	0.511627906976744 2	0.511627906976744 2	0.339460375000042	0.145010006999882 5	22912
42	StructOnly+LR	0.643658734451054 6	0.602127170337943 5	0.616279069767441 8	0.5875	0.635135135135135 1	0.546511627906976 7	1.060493466000025 4	0.098838553999939 9	19
42	WordTFIDF+LR	0.787182260681449 4	0.790507148986201 1	0.761627906976744 2	0.748466625766871 17	0.792207792207792 2	0.709302325581395 4	0.254508190000024 1	0.052071391000026 7	9310
42	WordTFIDF+SVM	0.806111411573823 6	0.794496467701207 5	0.779069767441860 5	0.768292682926829 3	0.807692307692307 7	0.732558139534883 7	0.194567487000085 7	0.038737083999876 6	9310
42	CharTFIDF+SVM	0.815575987020010 8	0.837887161871269 7	0.790697674418606	0.772151898734177 2	0.847222222222222 2	0.709302325581395 4	1.229104670999959 1	0.253823287000159 3	45919
42	CharTFIDF+Struct+SGD	0.621957815035154 1	0.590810151929920 5	0.581395348837209 3	0.612903225806451 6	0.57	0.662790697674418 6	1.731923173000041	0.248602272999960 4	45937

Table VI. Main results (mean and std across seeds; std=0.0 for single-seed evaluation).

model	auroc_mean	auroc_std	auprc_mean	auprc_std	f1 mean	f1_std	acc mean	acc std	trains_mean	infers_mean	n params
Char5gramLM	0.514196863169280 7	0.0	0.522999150716052 3	0.0	0.511627906976744 2	0.0	0.511627906976744 2	0.0	0.339460375000042	0.145010006999882 5	22912.0
CharTFIDF+SVM	0.815575987020010 8	0.0	0.837887161871269 7	0.0	0.772151898734177 2	0.0	0.790697674418606	0.0	1.229104670999959 1	0.253823287000159 3	45919.0

CharTFIDF+Struct+SGD	0.62195781503515 41	0.0	0.5908101519299205	0.0	0.6129032258064516	0.0	0.5813953488372093	0.0	1.73192317300041	0.2486022729999604	45937.0
HeuristicRisk	0.5324499729583558	0.0	0.5319781330903677	0.0	0.6666666666666666	0.0	0.5	0.0	0.0	0.0008319730000039	0.0
StructOnly+LR	0.6436587344510546	0.0	0.6021271703379435	0.0	0.5875	0.0	0.6162790697674418	0.0	1.06049346600000254	0.0983855399999399	19.0
WordTFIDF+LR	0.7871822606814494	0.0	0.7905071489862011	0.0	0.7484662576687117	0.0	0.7616279069767442	0.0	0.2545081900000241	0.0520713910000267	9310.0
WordTFIDF+SVM	0.8061114115738236	0.0	0.7944964677012075	0.0	0.7682926829268293	0.0	0.7790697674418605	0.0	0.1945674870000857	0.0387370839998766	9310.0

Table VII. Runtime and model size summary.

model	train_s_mean	infer_s_mean	n_params
Char5gramLM	0.3394603750000442	0.1450100069998825	22912.0
CharTFIDF+SVM	1.2291046709999591	0.2538232870001593	45919.0
CharTFIDF+Struct+SGD	1.731923173000041	0.2486022729999604	45937.0
HeuristicRisk	0.0	0.0008319730000039	0.0
StructOnly+LR	1.0604934660000254	0.0988385539999399	19.0
WordTFIDF+LR	0.2545081900000241	0.0520713910000267	9310.0
WordTFIDF+SVM	0.1945674870000857	0.0387370839998766	9310.0

Table IX. Error rates by mutation type for the best model.

mutation_type	n_bug	recall	mean_score
const_flip	35	0.8	0.0266694972071331
edge_flip	14	1.0	0.1270741281026801
eq_flip	13	0.9230769230769232	0.0330472644013491
logic_flip	8	0.25	-0.0259833890020868

nb_to_b	16	0.3125	-0.0255146745579064

Table X. Bootstrap confidence intervals for the best model.

metric	mean	ci2.5	ci97.5
AUROC	0.8151228892900307	0.7434495793340443	0.8803734942593638
AUPRC	0.8397613977636723	0.7662402454101089	0.9014549149000598

Table XI. Qualitative error examples (truncated snippets).

type	sample_id	mutation_type	score	text

FP	cvdp_copilot_16qam_mapper_0011::rtl/16qam_mapper.sv::21:51::clean	none	0.0917766592418029	2'b01: mapped_symbol_I[i] = -1; // MSBs 01 -> I = -1
				2'b10: mapped_symbol_I[i] = 1; // MSBs 10 -> I = 1
				2'b11: mapped_symbol_I[i] = 3; // MSBs 11 -> I = 3
				endcase
				case (symbol[1:0])
				2'b00: mapped_symbol_Q[i] = -3; // LSBs 00 -> Q = -3
				2'b01: mapped_symbol_Q[i] = -1; // L

FP	cvdp_copilot_cont_adder_0028::rtl/cont_adder_top.sv::15:45::clean	none	0.0813128843637432	output logic threshold_1, output logic threshold_2, output logic sum_ready); logic signed [DATA_WIDTH-1:0] sum_accum; logic [15:0] sample_count; logic signed [DATA_WIDTH-1:0] weighted_input; logic signed [DATA_WIDTH-1:0] new_sum;

				logic threshold_1_comb ;
FP	cvdp copilot 16qam mapper 0011::rtl/16qam mapper.sv::15:45::clean	none	0.0714791319745631	for (i = 0; i < N; i++) begin logic [IN WIDTH-1:0] symbol; assign symbol = bits[(i+1)*IN_WIDTH - 1 -: IN WIDTH]; always_comb begin case (symbol[3:2]) 2'b00: mapped symbol I [i] = -3; // MSBs 00 -> I = -3 2'b01: mapped_symbol_I [i] = -1; // MSBs 01 -> I = -1 2'b10: mapped symbol I [i] = 1; // MSBs 10 -> I
FP	cvdp copilot events to apb 0019::rtl/apb controller.sv::60:90::clean	none	0.0555615906906911	event list <= event_list & ~event sel; end end end always @(*) begin next state = current state; event_sel = event sel ff; case (current_state) IDLE: begin if (event list[0]) begin next state = SETUP; event sel = 3'b001;

				end else if
FP	cvdp_copilot_events to apb 0019::rtl/apb controller.sv::114:144::clean	none	0.0219517008618861	end always @(*) begin if (!reset_n) begin sel_addr_next = sel addr; sel_data_next = sel data; end else begin if (next state == SETUP || next state == ACCESS) begin case (event sel) 3'b001: begin sel addr next = addr_a_i; sel_data_next = d
FN	cvdp_copilot_binary multiplier 0016::rtl/binary multiplier.sv::0:30::bug::nb_to_b	nb_to_b	-0.1467997719188873	module binary multiplier #(parameter WIDTH = 32)(input logic clk, input logic rst_n, input logic [WIDTH-1:0] A, input logic [WIDTH-1:0] B, input logic valid_in, output logic [2*WIDTH-1:0] Product, output logic valid out); integer i; logic [2*WIDTH-1:0] sum; logic [WIDTH-

				1:0] cnt;

FN	cvdp_copilot_configurable_digital_low_pass_filter_0018::rtl/fsm_linear_reg.sv::15:45::bug::nb_to_b	nb_to_b	-0.1096927087051426	COMPUTE = 2'b01,
				DONE = 2'b10
				} state t;
				state t current_state, next state;
				logic signed [2*DATA_WIDTH-1:0] compute1;
				logic signed [DATA_WIDTH:0] compute2;
				always ff @(posedge clk or posedge reset) begin
				if (reset)
				
				current state = IDLE;
				else
				
				current_state <= next state;
				end
				always_comb begin
				case (curren

FN	cvdp_copilot_binary_multiplier_0016::rtl/binary_multiplier.sv::22:52::bug::logic_flip	logic_flip	-0.1085687481267499	done <= 0;
				start <= 0;
				A latched <= 0;
				B_latched <= 0;
				end else begin
				if (valid in) begin
				start <= 1;
				
				A_latched <= A;
				
				B latched <= B;
				end
				if (start || !done) begin
				if (cnt < WIDTH) begin

				if (A_latched[cnt])
FN	cvdp_copilot_dbi_0012::rtl/dbi_enc.sv::0:30::bug::const_flip	const_flip	-0.0970380991952052	module dbi_enc(input wire rst_n, // Asynchronous reset input wire clk, // Clock input wire [39:1] data_in, // Data input before DBI logic input wire dbi_enable, output wire [4:0] dbi_cntrl, // indicating when to enable and disable 'data bus inversion' operation. output wire [39:0] data_out // Data output after DBI logic); wire [39:0] next_db
FN	cvdp_copilot_events_to_apb_0019::rtl/apb_controller.sv::15:45::bug::logic_flip	logic_flip	-0.0920725836240563	output logic apb_pwrite_o, // APB write signal output logic [31:0] apb_pwdata_o, // 32-bit APB write data output input logic apb_pready_i // APB ready signal from the peripheral); typedef enum logic [1:0] { IDLE, SETUP, ACCESS } state_t; logic [3:0] count; logic [2:0] event_sel; logic [2:0] event_sel_ff;

Discussion

The experiments establish a quantitative baseline for TB-free RTL anomaly detection on real SystemVerilog code windows. Several insights follow directly from the empirical results.

Character n-grams provide strong signal for RTL anomaly ranking. The CharTFIDF+SVM baseline is the best-performing model and consistently outperforms word TF-IDF variants. RTL contains dense operator and punctuation patterns (e.g., <=, ==, &&, begin/end, and bit slicing), and many bug-like anomalies in this benchmark are single-token perturbations of those patterns. Character n-grams represent these low-level cues without requiring a full parser, which explains the strong AUROC and AUPRC results.

Structural statistics help but do not replace content. The structural-only logistic regressor achieves moderate AUROC, showing that anomalies correlate with code complexity and with the presence of sequential constructs. However, the gap to text models confirms that risk spotting depends on specific local content patterns rather than only on snippet shape.

Error modes identify where richer semantics are required. The best model detects edge-sensitivity and equality flips well, but recall is low on logic flip and nb_to_b. These two operators preserve many surrounding tokens and require understanding of control-flow intent (for && versus ||) and scheduling semantics (for <= versus =). This observation motivates extensions that incorporate higher-level representations: AST or data-flow features (as in GraphCodeBERT-style pretraining for software code) [20], width/type inference from SystemVerilog rules [8], and sequential semantics approximations.

Relationship to conventional lint and formal methods. Lint tools such as Verilator and Verible already identify many syntactic and stylistic issues [9], [10], [13]. Our TB-free anomaly task is complementary: the models rank snippets by statistical irregularity relative to a training corpus, even when the code is syntactically valid. Formal methods can prove or refute explicit assertions using bounded model checking or SAT-based techniques [14], [15], and SymbiYosys makes these flows accessible in open-source environments [11], [12]. TB-free anomaly scoring can be positioned earlier: it prioritizes where to write assertions or where to invest in formal and simulation effort.

Implications for LLM-assisted RTL review. LLM-based tools are increasingly evaluated on hardware benchmarks such as VerilogEval and CVDP [1], [4], [5]. Modern code models (BERT, CodeBERT, and their structural extensions) demonstrate that representation learning can capture rich syntax and semantics for software [17]-[20]. A practical RTL reviewer can combine TB-free risk ranking with explanation: a lightweight model identifies high-risk snippets quickly, and an LLM generates a human-readable review rationale and targeted suggestions. Our benchmark provides a controlled environment for measuring whether such hybrid systems improve ranking beyond classical n-gram baselines.

From an engineering perspective, the baseline results provide two concrete guidelines for system design.

First, the strongest baseline uses only linear classification over character n-grams. This means that a production TB-free reviewer can start with a very small and robust model that trains quickly, has a transparent feature space, and runs efficiently in CI. The model can be trained per project (to match local coding conventions) or trained on a pooled corpus (to support cross-project transfer). Because the model is linear, its top-weighted n-grams can be inspected to verify that the scorer responds to meaningful RTL tokens and operators.

Second, the mutation-specific error profile identifies where the reviewer must move beyond surface text. In particular, nb_to_b errors require sensitivity to scheduling and to the distinction between combinational and sequential intent, which is governed by the language and by project style rules [8]. A model that has access to even partial parsing (for example, recognizing always ff blocks and assignment contexts) can directly address this blind spot. Similarly, logic_flip errors require understanding of boolean intent, which is best captured by control-flow representations or by data-flow-aware embeddings.

Finally, the benchmark supports integration with LLM-based code review. A practical workflow is to use the lightweight scorer to pick the top-K risky snippets, and then ask an LLM to explain why each snippet is risky, propose targeted assertions, and suggest focused simulation stimuli. This division of labor leverages the high throughput of classical models and the explanatory capability of LLMs, while keeping the overall system grounded in a measurable ranking objective on a reproducible dataset.

Limitations. This study evaluates anomaly detection on single-token mutations. Although the operators are chosen to resemble common RTL defect mechanisms, they do not cover all real bug classes (e.g., missing resets, width truncation due to implicit casts, or multi-line state machine logic errors). The benchmark is also derived from a single corpus (CVDP non-agentic comprehension) and therefore reflects its coding styles and domains. These limitations are explicit properties of the current evaluation and are directly addressed by extending the mutation set, adding multi-mutation

samples, and incorporating additional RTL corpora such as large SoC repositories.

Threats to validity and interpretation. The benchmark is derived from a single corpus and uses mutations that are local and synthetic. These design decisions make the task measurable and reproducible, but they also constrain the conclusions. The reported AUROC and AUPRC values measure sensitivity to local perturbations in CVDP-style RTL, not the full space of functional bugs in industrial SoCs. The group-by-problem split controls for direct leakage of the same design across train and test, but it does not guarantee that coding conventions or boilerplate patterns are disjoint. Consequently, the results are interpreted as a baseline for the TB-free setting rather than as a claim of universal bug-detection accuracy.

Benchmark design choices that increase difficulty are straightforward to implement in this framework. First, the mutation operators can be extended to multi-token and multi-line perturbations (for example, deleting a reset clause, swapping two case items, or changing a bus width) while preserving syntactic validity. Second, the labeling can be strengthened by adding compiler checks: a mutated snippet can be discarded if it fails to parse in an open-source SystemVerilog front-end (e.g., Verible) [13], ensuring that the buggy class remains syntactically valid. Third, the dataset can be expanded by including additional RTL repositories such as OpenTitan [6], [7], which would directly test cross-project generalization.

Integration with LLM-based review. The baseline study also clarifies the role of LLMs in a practical review assistant. The models in Table VI already provide a fast, deterministic ranking function. An LLM can be placed downstream to explain and contextualize anomalies. For example, given a high-risk snippet, an LLM can summarize the control intent, identify the mutated operator, and propose a targeted test or assertion. This division of labor aligns with current research trends where LLMs are evaluated on code generation and reasoning tasks [1], [4], [5], and it reduces the requirement for an LLM to scan entire codebases. It also enables retrieval-augmented prompting: the anomaly model identifies a small set of candidate snippets and the LLM receives those snippets plus nearby context, maximizing relevance while controlling cost.

Deployment considerations. In a real DV workflow, the output of TB-free anomaly detection is a prioritized list, not a binary decision. Threshold selection should therefore be driven by review budget. Fig. 7 and Table VII show that the best model runs quickly enough to be integrated into continuous integration: it can score every RTL change and highlight diffs that introduce high-risk patterns. The error analysis in Table IX indicates which defect classes require additional semantic tooling; these findings can guide where to invest in further model development.

Operationalization in code review systems. A practical deployment attaches anomaly scores to the same artifacts that engineers already use for review: diffs, files, and change lists. Because the benchmark operates on local windows, a repository integration can compute a score for every window intersecting a changed hunk and then surface the maximum or top-k windows in the review UI. This approach produces a heatmap-like workflow: reviewers read the highest-risk regions first and then expand to surrounding context. The runtime results in Table VII show that this scoring step is fast enough to run on every commit in a continuous-integration pipeline, and it can be configured as a non-blocking advisory signal or as a gating threshold for critical branches.

Interpretability and trust. Early RTL review is a human activity and adoption depends on whether the signal is explainable. The linear baselines used in this paper are compatible with simple, deterministic explanations. For example, TF-IDF plus a linear classifier associates weights with character or word n-grams, which can be inspected to highlight local token patterns that drive a high-risk score. Even without full natural-language explanations, surfacing the top contributing spans improves reviewer efficiency by directing attention to the exact operators or literals that appear anomalous. This property is important for TB-free settings because it allows engineers to validate the alert by inspection without running simulation.

Interaction with lint, formal, and LLM assistants. A deployed flow composes tools rather than replacing them. Lint provides rule-based checks, anomaly scoring provides corpus-based irregularity ranking, and formal/simulation provide correctness evidence when assertions and tests exist. The score can therefore serve as a routing function: high-risk snippets trigger stricter lint configurations, trigger automatic generation of candidate assertions, or trigger an LLM-assisted explanation pass that produces review comments grounded in the code. Because benchmarks such as CVDP and VerilogEval evaluate LLMs on RTL generation and understanding [1], [4], [5], the combination of a fast scorer and an LLM explainer creates a measurable path from anomaly detection to end-to-end automated RTL review.

Responsible interpretation. The anomaly score is a prioritization signal, not a proof of defect. False positives are expected and are acceptable when the score is used to allocate attention. Conversely, false negatives occur for semantically subtle issues (Table IX), so the absence of an alert does not justify skipping review. The most effective use case therefore matches the motivation of this paper: reducing manual review load by filtering low-risk code and accelerating discovery of high-risk regions before large-scale simulation.

Conclusions

This paper presents a complete and reproducible study of TB-free RTL anomaly detection for early chip verification. Using the CVDP benchmark [1] as a source of real SystemVerilog RTL, we construct a balanced binary dataset of 1092 snippet-level samples by injecting single-token mutations that emulate five defect mechanisms. Under a strict group split by problem id, seven baselines are evaluated without using testbenches or assertions. The best baseline, CharTFIDF+SVM, achieves AUROC 0.816 and AUPRC 0.838 on a held-out test split, with bootstrap 95% AUROC CI [0.743, 0.880]. Error analysis shows high sensitivity to edge and equality flips and lower sensitivity to boolean-operator and nonblocking-to-blocking changes, indicating where semantic representations are required. Overall, the results quantify how much review-relevant signal exists in RTL text alone and provide a clear baseline for future LLM4EDA systems that aim to automate early RTL code review and bug-risk spotting.

References

[1] N. Pinckney, C. Batten, M. Liu, H. Ren, and B. Khailany, "Comprehensive Verilog Design Problems: A Next-Generation Benchmark Dataset for Evaluating Large Language Models and Agents on RTL Design and Verification," arXiv:2506.14074, 2025.

[2] AbiralArch, "hardware-cvdp-complete," Hugging Face Datasets, accessed Jan. 2026.

[3] NVlabs, "cvdp benchmark," GitHub repository, accessed Jan. 2026.

[4] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "VerilogEval: Evaluating Large Language Models for Verilog Code Generation," arXiv:2309.07544, 2023.

[5] N. Pinckney, C. Batten, M. Liu, H. Ren, and B. Khailany, "Revisiting VerilogEval: A Year of Improvements in Large-Language Models for Hardware Code Generation," arXiv:2408.11053, 2024.

[6] OpenTitan Project, "OpenTitan Documentation," accessed Jan. 2026.

[7] lowRISC, "opentitan," GitHub repository, accessed Jan. 2026.

[8] IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language, IEEE Std 1800-2017, 2017.

[9] Verilator Developers, "Verilator Documentation," accessed Jan. 2026.

[10] Verilator Developers, "verilator," GitHub repository, accessed Jan. 2026.

[11] C. Wolf and J. Glaser, "Yosys - A Free Verilog Synthesis Suite," in Proc. Austrochip, 2013.

[12] YosysHQ, "SymbiYosys (sby) Documentation," accessed Jan. 2026.

[13] Chips Alliance, "Verible: SystemVerilog Developer Tools," GitHub repository, accessed Jan. 2026.

[14] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in Proc. TACAS, 1999.

[15] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," Formal Methods in System Design, vol. 19, no. 1, pp. 7-34, 2001.

[16] A. Hindle, E. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the Naturalness of Software," in Proc. ICSE, 2012.

[17] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in Proc. NAACL-HLT, 2019.

[18] A. Vaswani et al., "Attention Is All You Need," in Proc. NeurIPS, 2017.

[19] 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," Findings of EMNLP, 2020.

[20] 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, "GraphCodeBERT: Pre-training Code Representations with Data Flow," in Proc. ICLR, 2021.

[21] F. Pedregosa et al., "Scikit-learn: Machine Learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825-2830, 2011.

[22] A. Paszke et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in Proc. NeurIPS, 2019.

[23] Xinzhuo Sun, Yifei Lu, and Jing Chen, “Controllable Long-Term User Memory for Multi-Session Dialogue: Confidence-Gated Writing, Time-Aware Retrieval-Augmented Generation, and Update/Forgetting”, JACS, vol. 3, no. 8, pp. 9–24, Aug. 2023, doi: 10.69987/JACS.2023.30802.

[24] Hanqi Zhang, “DriftGuard: Multi-Signal Drift Early Warning and Safe Re-Training/Rollback for CTR/CVR Models”, JACS, vol. 3, no. 7, pp. 24–40, Jul. 2023, doi: 10.69987/JACS.2023.30703.

[25] Xinzhuo Sun, Jing Chen, Binghua Zhou, and Meng-Ju Kuo, "ConRAG: Contradiction-Aware Retrieval-Augmented Generation under Multi-Source Conflicting Evidence", JACS, vol. 4, no. 7, pp. 50–64, Jul. 2024, doi: 10.69987/JACS.2024.40705.

[26] Hanqi Zhang, "Risk-Aware Budget-Constrained Auto-Bidding under First-Price RTB: A Distributional Constrained Deep Reinforcement Learning Framework", JACS, vol. 4, no. 6, pp. 30–47, Jun. 2024, doi: 10.69987/JACS.2024.40603.

[27] T. Shirakawa, Y. Li, Y. Wu, S. Qiu, Y. Li, M. Zhao, H. Iso, and M. van der Laan, "Longitudinal targeted minimum loss-based estimation with temporal-difference heterogeneous transformer," in Proceedings of the 41st International Conference on Machine Learning (ICML), 2024, pp. 45097–45113, Art. no. 1836.

[28] K. Xu, H. Zhou, H. Zheng, M. Zhu, and Q. Xin, "Intelligent classification and personalized recommendation of e-commerce products based on machine learning," Proceedings of the 6th International Conference on Computing and Data Science (ICCDS), 2024.

[29] Q. Xin, Z. Xu, L. Guo, F. Zhao, and B. Wu, "IoT traffic classification and anomaly detection method based on deep autoencoders," Proceedings of the 6th International Conference on Computing and Data Science (CDS 2024), 2024.

[30] B. Wang, Y. He, Z. Shui, Q. Xin, and H. Lei, "Predictive optimization of DDoS attack mitigation in distributed systems using machine learning," Proceedings of the 6th International Conference on Computing and Data Science (CDS 2024), 2024, pp. 89–94.

[31] Z. Ling, Q. Xin, Y. Lin, G. Su, and Z. Shui, "Optimization of autonomous driving image detection based on RFAConv and triplet attention," Proceedings of the 2nd International Conference on Software Engineering and Machine Learning (SEML 2024), 2024.