Agentic Programmable Waveform Analysis

Felix Roithmayr Daniel Große
Institute for Complex Systems, Johannes Kepler University Linz, Austria
felix.roithmayr@jku.at daniel.grosse@jku.at

***Abstract*—Waveform analysis is a cornerstone of *Electronic Design Automation* (EDA), enabling engineers to derive critical insights from simulation data. Recently, the *Waveform Analysis Language* (WAL) has been introduced to automate this process by allowing engineers to create and execute analysis programs directly on waveforms. However, the manual development of such programs remains a significant bottleneck, requiring both domain expertise and substantial effort.**

In this work, we propose an agentic framework that integrates advanced *Large Language Model*s (LLMs) with a structured architecture to automate the synthesis and execution of WAL programs from natural-language queries. By combining knowledge retrieval, program synthesis, and two-stage validation, the proposed framework promotes the generation of syntactically and semantically correct WAL programs while maintaining adaptability to diverse waveform analysis tasks.

To evaluate the framework, we benchmark it against golden reference WAL programs on a suite of representative waveform analysis tasks, comparing nine LLM backends and running an ablation study to isolate the contribution of each framework component. We then demonstrate its versatility and scalability with a cache profiling case study, where it calculates cache hit rates across several cache configurations without requiring new simulation runs. The framework's conversational, session-based design lets engineers refine an analysis and pose follow-up queries over the same loaded waveform.

I. Introduction

Waveforms are among the primary artifacts used for design understanding, verification and debugging in *Electronic Design Automation* (EDA). They capture the evolution of signals over simulation time together with structural information about the design hierarchy, making them indispensable for inspecting the behavior of *Hardware Description Language* (HDL) designs. In practice, engineers rely on waveforms to investigate functional circuit behavior, timing problems, performance bottlenecks, and unexpected interactions between hardware modules. This role is particularly important because verification remains one of the most resource-intensive parts of hardware development, accounting for 60-70% of engineering effort in chip projects [1], with debugging representing a substantial share of this effort [2].

Despite the importance of waveforms in hardware development, their analysis is still largely performed manually. In practice, engineers inspect simulation traces using waveform viewers and integrated debug environments, including open-source tools such as GTKWave [3], VaporView [4], and Surfer [5], as well as commercial environments such as Synopsys Verdi [6], Siemens Questa Visualizer [7], and Cadence Verisium Debug/SimVision [8]. These tools provide powerful interactive visualization and debugging capabilities, but recurring analysis tasks still often require engineers to manually navigate traces, select relevant signals, and interpret low-level signal changes.

Modern verification languages and methodologies, such as SystemVerilog and *Universal Verification Methodology* (UVM), provide assertions, coverage collection, constrained-random stimulus generation, and reusable verification components [9]. However, they do not directly address the problem of generic, user-programmable post-simulation waveform analysis. As a result, answering recurring analysis questions, such as identifying protocol transactions, measuring request-response latency, computing throughput, or determining the root cause of a problem, often remains repetitive, time-consuming, and error-prone.

To overcome this limitation, the open-source *Waveform Analysis Language* (WAL) was introduced as a *Domain Specific Language* (DSL) for programmable waveform analysis [10], [11]. Rather than merely viewing waveforms, WAL allows engineers to create and execute analysis programs directly on waveform data. In WAL, waveform signals are exposed as program variables, while simulation time and design hierarchy are treated as core language concepts. This enables engineers to automate analyses that would otherwise require tedious manual inspection. In contrast to assertions, which primarily evaluate whether specified temporal properties hold or fail, WAL programs can perform arbitrary computations over waveform data, including event correlation, metric computation, statistical aggregation, and the extraction of higher-level information from low-level signal changes.

However, while WAL significantly enhances the automation of waveform analysis, engineers must still manually develop WAL programs tailored to their specific needs. This process involves understanding the design, identifying relevant signals and temporal relationships, and translating the intended analysis into correct WAL code. As a result, the manual development of WAL programs can be time-consuming and expertise-intensive, particularly for complex analyses. The recent advancements in generative AI offer a promising solution to bridge this gap.

In this paper, we address the challenges of manual waveform analysis by introducing an agentic framework that integrates WAL with natural-language-driven program synthesis. The proposed framework leverages *Large Language Model*s (LLMs) within an agentic architecture to interpret user queries, coordinate program synthesis, validate correctness through syntax and semantic checks, and execute the resulting analysis programs on waveform data. Due to the size and the nature of raw data in real-world waveform files, any angentic analysis needs to incorporate deterministic steps to perform the actual

data processing. LLMs struggle to process large datasets deterministically, but enabling LLMs to use a DSL is a powerful tool to overcome this limitation. By automating the translation of user queries into executable WAL programs, the proposed framework provides a structured methodology for waveform analysis, reducing manual effort and improving the efficiency of recurring analysis tasks.

To evaluate the proposed framework, we conduct two types of experiments. First, we perform a golden reference evaluation on a suite of representative waveform analysis tasks, where the framework synthesizes WAL programs from natural-language queries. We benchmark their correctness against ground-truth references across nine LLM backends and run an ablation study to isolate the contribution of each framework component. Second, we demonstrate the framework's versatility and scalability through a cache profiling case study, where it analyzes data memory access patterns of an IBEX RISC-V core [12] running the nettle-AES benchmark. Here, the framework synthesizes and executes WAL programs to estimate cache hit rates for varying cache sizes and configurations, a conversational what-if analysis of cache performance that needs no additional simulation runs.

II. Related Work

Surveys document a broad application of LLMs to EDA, spanning hardware design, HDL code generation, verification, debugging, and design analysis [13]. In design, LLMs have been evaluated for generating *Register Transfer Level* (RTL) from natural-language specifications [14] and for benchmarking Verilog-generation capabilities [15]. In verification, recent work targets automatic testbench creation [16] and assertion synthesis from design specifications [17]. In security verification, recent work has also explored the automatic generation of security assertions and information-flow properties, including LLM-based generation of SystemVerilog assertions [18]–[20] and information-flow-based specification mining for hardware security property generation [21], [22]. In debug and correction, machine-learning techniques have been applied to post-silicon bug localization [23], while LLM-based approaches address hardware debugging [24], syntax-error repair using compiler feedback [25], and iterative correction of generated RTL using simulator and waveform-tracing feedback [26]. These directions demonstrate the feasibility and potential impact of LLMs in EDA flows. In contrast, this work focuses on translating natural-language analysis queries into executable WAL programs for waveform analysis.

Protocol checkers and transaction monitors exist for standard on-chip buses such as AXI. For example, prior work proposes rule-based AXI protocol checkers [27] and reconfigurable bus monitors for protocol and performance monitoring [28], [29]. These approaches are effective for predefined compliance checks and metrics, but they are not designed for on-demand, task-specific waveform analyses derived from natural-language intents. Similarly, trace-based performance and cache analysis has a long history, including trace-driven memory simulation [30] and cache profiling [31]. However, such analyses are typically implemented as specialized tools or workload-specific analysis flows, whereas our work targets the automatic generation of executable waveform-analysis programs for user-specified tasks.

III. WAL Background

The *Waveform Analysis Language* (WAL) is a DSL for programmable waveform analysis [10], [11], [32]. Prior work has demonstrated WAL's expressiveness on a range of advanced waveform-analysis tasks, including protocol and pipeline analysis [33], processor and cache performance characterization [34], and pre-silicon power side-channel assessment [35]. Further applications include compiler-integrated waveform analysis [36], interactive debugging with virtual signals [37], and filtering circuit-optimization rules before formal verification [38]. This makes WAL a suitable target language for synthesizing executable waveform-analysis programs from natural-language queries. In this work, we use the publicly available WAL reference implementation [39] to execute the generated programs.

A WAL program is evaluated over a loaded waveform trace. WAL uses a compact, Lisp-inspired prefix syntax in which expressions are written as `(op a b c ...)`, with the operator or function `op` appearing first, followed by its whitespace-separated arguments. Once a waveform has been loaded, signal names can be used directly as variables in WAL expressions. A signal reference evaluates to that signal's value at the current simulation time, avoiding the need for explicit waveform-access code.

Simulation time and design hierarchy are first-class concepts in WAL. Relative-time access is provided by the `@` operator; for example, `x@-1` evaluates signal `x` at the previous time point. Built-in traversal operations evaluate expressions across a trace and collect matching time points. For example, `(find (= req ack 1))` traverses the waveform and returns every time point at which both `req` and `ack` have the value `1`.

IV. Agentic Framework

We first present the framework's architecture (Section IV-A) and then detail each major step using a running example (Sections IV-B through IV-F).

A. Architecture

Fig. 1 depicts the architecture, whose six steps are labeled ❶–❻. The operator agent is a natural-language-driven coordinator that interprets user queries, drives all components, and reports results. In Step ❶ *Waveform Retrieval*, it calls the *Wave MCP* to load the waveform and read its metadata, notably the signal names in the design hierarchy. In ❷ *Add Documentation*, user-supplied domain- and design-specific documentation is added to the *knowledge base*, which also holds the WAL reference and curated examples. In ❸ *WAL generation*, the operator agent invokes the WAL synthesis agent: passing the signal list, general problem context, and a crafted prompt, which retrieves relevant examples from the *knowledge base* and returns a *candidate program*.

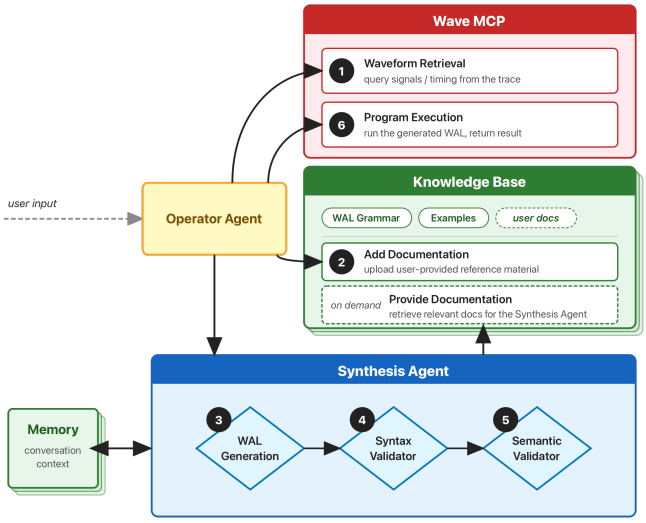


Fig. 1: Overview of the *Agentic Framework*

This candidate is then filtered by two deterministic checks: ❹ *Syntax Validator* verifies it against the WAL grammar, and ❺ *Semantic Validator* checks semantic correctness (e.g., that referenced signals exist). Finally, in ❻ *Program Execution*, the operator agent invokes the *Wave MCP* to run the *WAL program*; a satisfactory result is reported to the user, otherwise the operator agent refines the analysis or re-synthesizes.

B. Operator Agent

The operator agent coordinates the framework: it interprets the analysis intent behind a natural-language query and plans the invocation order of components from a skill [40]. We use a small running example throughout this section.

Example 1. *The user issues the query*

```
Analyze the trace file axibus-trace.fst and count the
total number of AXI write burst transactions. Return ONLY
the integer count.
```

and attaches AXI protocol documentation. The operator agent derives two goals: load `axibus-trace.fst` *and inspect it for AXI signals, and synthesize a WAL program that counts write bursts over the full trace. It then assembles a synthesis prompt from two context sources: (1) the signal list and metadata from the* Wave MCP *(e.g., signals matching* `axi_*`*), and (2) the user's protocol context (e.g., an AXI4 burst ends with* `wlast`*). This prompt is the primary means by which the operator agent steers the WAL synthesis agent toward a correct candidate.*

Because it owns the overall goal, the operator agent may invoke the pipeline several times and interpret the results as the user's intent requires.

C. Wave MCP: Waveform Retrieval

The *Wave MCP* mediates all interaction between the agents and the waveform data: it wraps the WAL runtime and exposes it as tools over the *Model Context Protocol* (MCP), decoupling the agents from the backend so the same framework can target different waveform sources unchanged. Its inspection tools let the operator agent enumerate signals in the design hierarchy, filter them by regular expression, and read metadata such as the signal count and time range. Invocation of this subsystem is done in Step ❶, to ground synthesis in the waveform's actual signal inventory.

Example 2. *In the running example, the operator agent loads* `axibus-trace.fst` *and filters the signal list with* `axi_*`*, identifying the channel signals relevant to burst detection (*`wvalid`*,* `wready`*,* `wlast`*, ...).*

The Wave MCP is stateful: it loads waveform metadata once and retains access across tool calls within a session, avoiding redundant I/O and enabling the conversational analysis pattern in which the user poses several queries against the same waveform.

D. Knowledge Base

The *knowledge base* is the *Retrieval-Augmented Generation* (RAG) store for the WAL synthesis agent. It is populated from three sources: (1) the official WAL language specification (grammar and semantics); (2) the standard-library documentation (built-in functions and operators); and (3) example programs illustrating idiomatic usage. Users may additionally supply design-specific documentation in Step ❷, such as protocol descriptions or naming conventions.

Example 3. *For our example, the user has uploaded the AXI protocol description, so the* knowledge base *stores it alongside the standard WAL documentation and examples.*

E. WAL Synthesis Agent

The WAL synthesis agent is a dedicated agent whose sole job is to synthesize WAL programs. Invoked in Steps ❸-❺ with a synthesis prompt, it produces a *candidate program*.

Example 4. *For the running example, the first candidate is shown in Listing 1: Lines 1–4 alias the design-hierarchy signals, Line 5 defines a counter, Line 6 iterates over all timesteps, Line 7–9 increment the counter when a burst-end condition holds on a rising clock edge, and Line 10 prints the result.*

Delegating synthesis to a dedicated agent lets its model, temperature, and prompt be tuned independently of the orchestration logic, keeps the operator agent's context focused on coordination, and lets the WAL synthesis agent retain context across attempts to improve retries after failure. In Step ❸, it queries the *knowledge base* with a retrieval prompt derived from the operator agent's prompt and the signal context; the top chunks supply the examples and reference material that steer it toward correct programs.

```
1  (alias clk "TOP.tb.clk")
2  (alias wvalid "TOP.tb.M_Axi_WValid")
3  (alias wready "TOP.tb.M_Axi_WReady")
4  (alias wlast "TOP.tb.M_Axi_WLast")
5  (define cnt 0)
6  (while (step 1)
7    (when (&& clk (! clk@-1))
8        (when (&& wvalid wready wlast)
9              (set! cnt (+ cnt 1))))
10 (printf "Total␣AXI␣write␣burst␣transactions:␣%d\n" cnt)
```

Listing 1: First candidate WAL program for running example

Example 5. *In the running example, retrieval surfaces (a) a chunk describing WAL's edge-counting construct, (b) the WAL grammar reference, and (c) the AXI protocol condition marking burst completion.*

Before a candidate is returned, it passes two checks; any errors are sent back to the operator agent together with the broken candidate so it can re-prompt the WAL synthesis agent.

❹ *Syntax Validation*: The candidate is first checked against the WAL grammar by a deterministic parser, which yields exact error messages (line number, expected and actual token), far more actionable for re-synthesis than the heuristic feedback of an LLM-based check.

Example 6. *Here, the first candidate (Listing 1) omits a closing parenthesis in Line 9;* syntax validation *reports the offending line and expected token, and re-synthesis repairs it.*

❺ *Semantic Validation*: A candidate that passes *syntax validation* is then checked for semantic correctness (e.g., that referenced signals exist and that operations receive operands of the expected type). Unlike *syntax validation*, this requires waveform metadata, so it runs in a standalone parser with access to it. Together, the two form a filter that catches structural and logical errors before any execution.

Example 7. *For our example,* semantic validation *confirms that* `wvalid`*,* `wready`*, and* `wlast` *resolve to actual single-bit signals within* `axibus-trace.fst`*.*

F. Wave MCP: WAL Execution

Once a candidate passes both stages, the operator agent invokes the *Wave MCP* to execute it against the loaded waveform in Step ❻, returning the programś output. If execution fails (e.g., due to a runtime error not caught by validation) the operator agent captures the error and may return to Step ❸ for another attempt, using the same error-aware re-prompting as before; otherwise it formats the result and presents it to the user.

Example 8. *The final program is that of Listing 1 with the missing parenthesis added; it returns the scalar* `42`*, and the operator agent simply reports "42"; since in Example 1 the user specified to ONLY return the integer count.*

A successful execution need not end the interaction. Because the operator agent keeps the full conversational state and the waveform stays loaded in the *Wave MCP*, the user can pose follow-ups without reloading the waveform or re-ingesting documentation.

Example 9. *The user might follow up with "*of those, how many exceed 8 beats?*", reusing the already-loaded waveform and the augmented knowledge base.*

V. Golden-Output Evaluation

To evaluate the framework's ability to correctly translate natural-language analysis queries into WAL programs, we execute each synthesized program on the corresponding waveform data and compare its output against a golden-reference output. The golden-reference outputs are produced by hand-written WAL programs executed on the same waveform data and serve as ground truth. We limit this evaluation to tasks for which a correct WAL reference program can be written by hand with reasonable effort.

A. Experimental Setup

All experiments were conducted on an Apple M4 Pro with 24 GB of unified memory. Open-weight models were served locally on this machine, while commercial models were accessed through their respective cloud APIs.

Given the interactive nature of the intended use case, we impose a 5-minute timeout on each task instance. This value was chosen to reflect a practical upper bound for interactive use, informed by empirical testing showing that correctly solved tasks converge well within this window for the task complexity considered here, while still allowing sufficient margin for valid solutions that take slightly longer. Naturally, this bound would need to be revisited for substantially larger or more complex tasks, where longer execution times are to be expected.

B. Agentic Framework Configuration

1) Operator Agent: We use Claude Opus 4.6 [41] as the operator agent. The operator agent is held constant across all runs, so the reported per-model numbers reflect synthesis quality under a fixed orchestration policy.

2) Synthesis Agents: We instantiate the *WAL Synthesis Agent* with nine different LLM models, covering both locally hosted open-weight models and commercial models. The open-weight models are GPT-OSS-20B [42], Llama-3-8B [43], and DeepSeek-R1-14B [44]. The commercial models include Gemini-2.5-Flash [45], Gemini-2.5-Pro [46], Gemini-3-Flash-Preview [47], Gemini-3.1-Pro-Preview [48], Claude Sonnet 4.6 [49], and Claude Opus 4.6 [41]. No prompt tuning was performed for any of these models.

3) Baselines: We compare against two baselines: *base* uses Claude Code with Opus 4.6 directly on the same user-provided analysis queries, without access to any components of the proposed framework; *base+lib* uses the same setup, but additionally provides the pylibfst Python library and references it in the system prompt, since all traces are stored in the FST format. This baseline tests whether giving the model access to a suitable waveform-parsing library is sufficient to close the gap to the proposed framework. Both baselines are subject to the same timeout as the proposed framework to ensure a fair comparison.

C. Analysis Tasks

In the following, we provide a *high-level summary* of the analysis tasks used for the evaluation. Some tasks are evaluated on particularly large traces that reach up to 240 MB in compressed FST format, corresponding to approximately 16 GB of equivalent waveform data when converted to VCD, and span up to 110 million clock cycles. These trace sizes were chosen to bridge the gap between small, simple tests and the full-scale simulations encountered in industrial verification flows. We label *Direct Value Tasks* as Di, *Large Direct Value Tasks* as Li, and *Qualitative Tasks* as Qi, respectively:

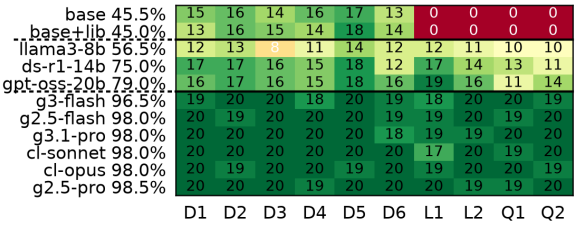


Fig. 2: Heatmap of the results for the analysis tasks. Each cell shows the number of successful runs out of 20; row labels also report aggregate accuracy.

D1) Count the total number of AXI write bursts that completed.
D2) Get AWLEN value during the first write address handshake.
D3) Get BRESP value for the last write response.
D4) Get the total number of write data beats.
D5) Get the total number of AXI write transactions that completed.
D6) Get AWADDR value for the first write address handshake.
L1) Count the number of flipped bits on every rising clock compared to the previous one.
L2) Find the three most accessed memory addresses via the SPI controller and print their number of reads and writes.
Q1) Verify AXI write channel ordering.
Q2) Verify that WLAST is asserted on the last beat of each burst.

The *Qualitative Tasks* require additional explanation. Unlike the *Direct Value Tasks*, they are not evaluated by exact-value matching; instead, a run passes if its output contains at least one from a set of required facts and none from a set of forbidden facts. These tasks were evaluated on traces of similar size to those used for the *Large Direct Value Tasks*.

Each task is evaluated on 4 different waveforms, with 5 independent framework runs per waveform, resulting in 20 runs per task and per WAL synthesis agent. These repeated runs account for the non-determinism inherent to LLM-based synthesis: the same analysis query and waveform can yield syntactically different but equally valid WAL programs across runs, and some runs may succeed where others fail. For Qi, this check is performed by DeepEval [50] using an LLM-as-a-judge metric, to end up with the same binary pass/fail per run as the Di and Li tasks.

D. Result Discussion

Fig. 2 presents the evaluation results as a heatmap, with one row per evaluated configuration and one column per analysis task (cf. Section V-C). Each cell reports the number of successful runs out of 20, where success means that the produced output by a WAL program matched the golden reference. The row label also reports the aggregate accuracy across all tasks. The two baselines are shown first, followed by the open-weight models and then the commercial models; within the model groups, rows are ordered by increasing aggregate accuracy. The color scale highlights the same information visually, with red cells indicating low success counts and green cells indicating high success counts.

As an example, consider the baseline row for task $D1$: 15 of the 20 runs succeeded (recall 4 different waveforms $\times$ 5 independent runs per waveform). Summing the successful runs across all ten tasks gives $15 + 16 + \cdots + 0 + 0 = 91$, yielding an overall baseline accuracy of $91/200 = 45.5\%$, as shown in the row label. *base+lib* follows a similar pattern but performs slightly worse overall: On smaller traces, Claude Code with Opus 4.6 often succeeds by converting the FST file to VCD and parsing the resulting text directly. Providing pylibfst appears to shift the model toward a more code-heavy parsing strategy using python, which more frequently fails to complete within the timeout.

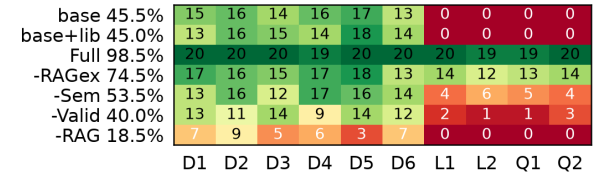


Fig. 3: Heatmap of the ablation-study results. Each cell shows the number of successful runs out of 20; row labels report aggregate accuracy.

Several trends are visible in the heatmap. First, all configurations using the proposed framework outperform both baselines in terms of overall accuracy. The baselines achieve nonzero success only on the direct-value tasks Di and fail completely on the larger and qualitative tasks Li and Qi, indicating that direct prompting alone does not scale to larger waveform-analysis settings. Second, among the framework-based configurations, the open-weight models perform reasonably well but trail the commercial models. Their lower success counts are most pronounced on the Li and Qi tasks, reflecting the added difficulty of large traces and qualitative output requirements compared with the direct-value tasks. Third, the commercial models achieve the highest accuracies overall, with several configurations reaching 98% or higher. This gap between open-weight and commercial models is likely due to a combination of larger context windows, stronger coding ability, and better instruction following, all of which increase the likelihood of synthesizing a WAL program that passes our two-stage validation and produces the correct result. Overall, the results show that the proposed framework is decisive for robust waveform analysis, while model capability still has a measurable effect on final task accuracy.

To identify which framework components contribute most to the observed gains over the baselines, we next perform an ablation study.

E. Ablation Study

To identify which framework components contribute most to the observed accuracy gains, we perform a targeted ablation study using Gemini-2.5-Pro, the best-performing WAL synthesis agent in Fig. 2. The ablated variants are evaluated with the same analysis task set as before and are compared against the full framework as well as the two baselines introduced above.

Fig. 3 shows that each framework component contributes measurably to the final accuracy. Removing the retrieved WAL-specific examples (*-RAGex*) causes the smallest degradation among the ablations, reducing accuracy from 98.5% to 74.5%. This indicates that the model can still synthesize many correct programs from the remaining context, but that concrete examples are important for steering it toward the intended WAL idioms and coding patterns.

Removing semantic validation (*-Sem*) has a larger effect, reducing accuracy to 53.5%. The degradation is especially

visible on the large-trace and qualitative tasks, where incorrect programs may remain syntactically valid but only fail during execution or produce subtly incorrect outputs. In these cases, the absence of semantic validation prevents the framework from detecting and correcting errors before expensive waveform processing begins. Removing validation more broadly (*-Valid*), i.e., disabling both syntax and semantic validation, further reduces accuracy to 40.0%, confirming that validation is a central part of the synthesis loop.

The largest drop occurs when retrieval is removed entirely (*-RAG*), reducing accuracy to 18.5%. Without retrieved documentation or examples, the model frequently lacks the information about existing functions and operators needed to construct executable WAL programs, leading to failures even on several direct-value tasks. This result shows that RAG is the most critical component in the evaluated configuration. Overall, the ablation study confirms that the framework's performance is not due to the underlying model alone: retrieval provides the necessary task- and function-specific context, examples guide the model toward effective WAL usage, and validation prevents many incorrect programs from reaching final execution.

VI. Case Study: Analysis on IBEX Core

We now present a case study that goes beyond the tasks in Section V-C. Many real analyses carry state across an entire trace or refine a question over several conversational turns. To show the framework handles these, we profile data-memory access patterns of an IBEX RISC-V core [12] running the *nettle-AES* benchmark from Embench [51], targeting a cache-sizing question: what fully-associative *Least Recently Used* (LRU) data-cache size reaches a target hit rate for a given access stream?

The user asks:

```
Analyze the data memory accesses in ibex_trace.fst and
estimate the cache hit rate I would get for data cache
sizes of 1KB, 2KB, 4KB, and 8KB, assuming a 32-byte cache
line and a fully-associative LRU replacement policy.
```

The operator agent locates the IBEX data-interface signals (`data_req`, `data_gnt`, `data_addr[31:0]`, and the clock) and hands them to the WAL synthesis agent along with the query. The generated WAL program (omitted for space) tracks four independent LRU caches in a single pass over the trace: on each data-interface handshake it computes the cache-line address, checks each of the four lists for a hit or miss, updates the list accordingly, and prints one CSV row per cache size once the trace ends. The program passes syntax and semantic validation on the first attempt, and semantic validation confirms that all four signals resolve to their expected widths in `ibex_trace.fst`.

Parsing the resulting CSV, the operator agent reports a 46.7% hit rate at 1 KB. This is low because the AES implementation depends on both the standard S-box [52] and a 4 KB round-speedup lookup table [51], so a 1 KB cache cannot hold much of the working set. Hit rate climbs to 59.3% at 2 KB (+12.6 pp) and then to 88.8% at 4 KB (+29.5 pp, a 72% cut in misses), which is roughly where the AES working set starts

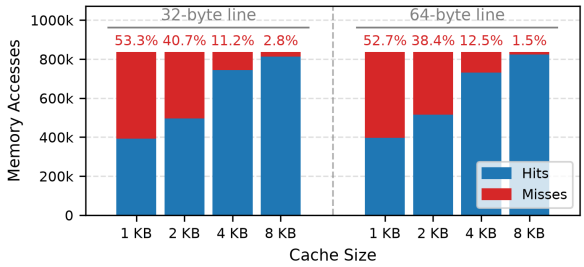


Fig. 4: Data-cache hit rate vs. cache size for the IBEX nettle-AES trace.

fitting in cache. At 8 KB it reaches 97.2%. Given the size of the jump between 2 KB and 4 KB and the diminishing returns past that, 4 KB looks like the right size for this workload. The waveform session is still open, so the user asks a follow-up:

```
Re-run the analysis with a 64-byte cache line instead of
32 bytes and see how the hit rates change. The results for
both queries should be plotted, split into two separately
labeled bar plots in a single figure.
```

The operator agent passes the existing script back to the WAL synthesis agent with the requested changes: line size set to 64, and the capacity list halved so the same four byte sizes are covered. This is a small edit to the earlier program, and synthesis, validation, and execution again finish in one pass. Fig. 4 shows both runs, as produced directly by the operator agent. The 64-byte line does slightly better than the 32-byte line at every size except 4 KB, but the gap is small enough that 4 KB with a 32-byte line still looks like the better choice overall.

We checked both generated WAL scripts and the results after the run. The program tracks four cache configurations over tens of thousands of trace events in a single pass. The follow-up reused that program with a small edit rather than re-synthesizing from scratch, illustrating how stateful sessions enable incremental refinement.

VII. Conclusions

In this paper, we presented an agentic framework for natural-language-driven waveform analysis that integrates WAL with LLMs through knowledge retrieval, program synthesis, and two-stage validation. The framework changes the role of WAL in the analysis workflow: instead of requiring engineers to manually write task-specific analysis programs, WAL becomes the executable backend for analyses specified in natural language. This allows engineers to focus on what metric, property, or behavior they want to analyze rather than how to implement the corresponding analysis.

Our evaluation demonstrates that this structured approach enables robust waveform analysis beyond simple direct-value queries. The ablation study further confirms that the framework's accuracy is not due to the LLM alone, but depends on the combination of retrieval, examples, and validation. Finally, the IBEX cache-profiling case study demonstrates interactive, stateful analysis of cache-design tradeoffs without requiring additional simulation runs.

Overall, the proposed framework moves waveform analysis from manually constructed WAL programs to on-demand, validated analyses synthesized from natural-language intent.

REFERENCES

[1] H. Foster, "Why first-silicon success is getting harder for system companies," 2025, verification Horizons, Siemens Digital Industries Software. [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons/2025/09/03/why-first-silicon-success-is-getting-harder-for-system-companies/.

[2] ——, "Part 4: The 2020 wilson research group functional verification study," 2020, verification Horizons, Siemens Digital Industries Software. [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons/2020/12/02/part-4-the-2020-wilson-research-group-functional-verification-study/.

[3] *GTKWave 3.3 Wave Analyzer User's Guide*, GTKWave, 2025, available online: https://gtkwave.sourceforge.net/gtkwave.pdf.

[4] L. Ramseyer, "VaporView: Waveform viewer extension for VS Code," 2024, available online: https://github.com/Lramseyer/vaporview.

[5] F. Skarman, L. Klemmer, D. Große, O. Gustafsson, and K. Laeufer, "Surfer – an extensible waveform viewer," in *CAV*, 2025, pp. 392–404.

[6] *Verdi Automated Debug System*, Synopsys, 2024, product datasheet. Available online: https://www.synopsys.com/content/dam/synopsys/verification/datasheets/verdi-ds.pdf.

[7] *Questa Visualizer Debug Environment*, Siemens EDA, 2025, product documentation. Available online: https://www.siemens.com/en-us/products/ic/questa-one/debug-analysis/visualizer-debug/.

[8] *Verisium Debug*, Cadence Design Systems, 2025, product documentation. Available online: https://www.cadence.com/en_US/home/tools/system-design-and-verification/ai-driven-verification/verisium-debug.html.

[9] *IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language*, IEEE Std. IEEE Std 1800-2023, 2024.

[10] L. Klemmer and D. Große, "WAL: a novel waveform analysis language for advanced design understanding and debugging," in *ASP-DAC*, 2022, pp. 358–364.

[11] ——, "WAVING goodbye to manual waveform analysis in HDL design with WAL," *TCAD*, vol. 43, no. 10, pp. 3198–3211, 2024.

[12] lowRISC contributors, "Ibex RISC-V Core," https://github.com/lowRISC/ibex, 2026.

[13] J. Pan, G. Zhou, C.-C. Chang, I. Jacobson, J. Hu, and Y. Chen, "A survey of research in large language models for electronic design automation," *TODAES*, vol. 30, no. 3, pp. 1–21, 2025.

[14] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg, "Benchmarking large language models for automated verilog RTL code generation," in *DATE*, 2023, pp. 1–6.

[15] Y. Lu, S. Liu, Q. Zhang, S. Wu, C. Yuan, and Q. Xu, "RTLLM: An open-source benchmark for design RTL generation with large language model," in *ASP-DAC*, 2024, pp. 722–727.

[16] R. Qiu, G. L. Zhang, R. Drechsler, U. Schlichtmann, and B. Li, "Autobench: Automatic testbench generation and evaluation using LLMs for HDL design," in *MLCAD*, 2024.

[17] Z. Yan, W. Fang, M. Li, M. Li, S. Liu, Z. Xie, and H. Zhang, "AssertLLM: Generating hardware verification assertions from design specifications via multi-LLMs," in *ASP-DAC*, 2025, pp. 614–621.

[18] R. Kande, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and J. Rajendran, "(Security) Assertions by Large Language Models," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 4374–4389, 2024.

[19] A. Menon, S. S. Miftah, A. Srivastava, S. Kundu, S. Kundu, A. Raha, S. Banerjee, D. Mathaikutty, and K. Basu, "OpenAssert: Towards secure assertion generation using large language models," in *VTS*, 2025, pp. 1–5.

[20] W. Fu, Y. Wang, Z. Lu, X. Guo, and G. Qu, "HADA: Leveraging multi-source data to train large language models for hardware security assertion generation," in *MLCAD*, 2025, pp. 1–7.

[21] C. Deutschbein, A. Meza, F. Restuccia, M. Gregoire, R. Kastner, and C. Sturton, "Toward hardware security property generation at scale," *IEEE Security & Privacy*, vol. 20, no. 3, pp. 43–51, 2022.

[22] C. Deutschbein, A. Meza, F. Restuccia, R. Kastner, and C. Sturton, "Isadora: Automated information-flow property generation for hardware security verification," *JCEN*, 2022.

[23] E. E. Mandouh and A. G. Wassal, "Application of machine learning techniques in post-silicon debugging and bug localization," *Journal of Electronic Testing*, vol. 34, no. 2, pp. 163–181, 2018.

[24] W. Fu, K. Yang, R. G. Dutta, X. Guo, and G. Qu, "LLM4SecHW: Leveraging domain-specific large language model for hardware debugging," in *AsianHOST*, 2023, pp. 1–6.

[25] Y.-D. Tsai, M. Liu, and H. Ren, "RTLFixer: Automatically fixing RTL syntax errors with large language models," in *DAC*, 2024, pp. 1–6.

[26] C.-T. Ho, H. Ren, and B. Khailany, "VerilogCoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (AST)-based waveform tracing tool," in *AAAI*, vol. 39, no. 1, 2025, pp. 300–307.

[27] C.-H. Chen, J.-C. Ju, and I.-J. Huang, "A synthesizable AXI protocol checker for SoC integration," in *ISOCC*, 2010, pp. 103–106.

[28] P.-C. Lee and I.-J. Huang, "Reconfigurable bus monitor tool suite for on-chip SoC for performance and protocol monitoring," in *ReCoSoC*, 2014, pp. 1–6.

[29] H.-M. Kyung, T.-H. Lee, S.-H. Lee, H.-G. Lee, S.-J. Song, J.-H. Ahn, S.-B. Park, and C.-M. Kyung, "Performance monitor unit design for an AXI-based multi-core SoC platform," in *ACM SAC*, 2007, pp. 1565–1572.

[30] R. A. Uhlig and T. N. Mudge, "Trace-driven memory simulation: A survey," *ACM Computing Surveys*, vol. 29, no. 2, pp. 128–170, 1997.

[31] A. R. Lebeck and D. A. Wood, "Cache profiling and the SPEC benchmarks: A case study," *Computer*, vol. 27, no. 10, pp. 15–26, 1994.

[32] L. Klemmer and D. Große, *Versatile Hardware Analysis Techniques – From Waveform-based Analysis to Formal Verification*. Springer, 2025.

[33] ——, "Waveform-based performance analysis of RISC-V processors: late breaking results," in *DAC*, 2022, pp. 1404–1405.

[34] ——, "An extensible and flexible methodology for analyzing the cache performance of hardware designs," in *FDL*, 2024, pp. 1–8.

[35] A. Nešković, A. Kabour, M. Eltantawi, M. Berekovic, R. Meyer, and S. Mulhem, "SCA-WAL: An open-source framework for power side channel assessment at RTL and netlist level," *IEEE ESL*, 2025.

[36] F. Skarman, L. Klemmer, O. Gustafsson, and D. Große, "Enhancing compiler-driven HDL design with automatic waveform analysis," in *FDL*, 2023, pp. 1–8.

[37] L. Klemmer and D. Große, "Towards a highly interactive design-debug-verification cycle," in *ASP-DAC*, 2024, pp. 692–697.

[38] D. Große, L. Klemmer, and D. Bonora, "Using formal verification methods for optimization of circuits under external constraints," in *DATE*, 2024, pp. 1–6.

[39] "WAL: the Waveform Analysis Language," https://github.com/ics-jku/wal.

[40] Anthropic, "The complete guide to building skills for Claude," https://resources.anthropic.com/hubfs/The-Complete-Guide-to-Building-Skill-for-Claude.pdf, 2026.

[41] ——, "Claude opus 4.6 system card," 2026. [Online]. Available: https://www-cdn.anthropic.com/6a5fa276ac68b9aeb0c8b6af5fa36326e0e166dd.pdf

[42] OpenAI *et al.*, "gpt-oss-120b & gpt-oss-20b model card," 2025. [Online]. Available: https://arxiv.org/abs/2508.10925

[43] AI@Meta, "Llama 3 model card," 2024. [Online]. Available: https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md

[44] DeepSeek-AI, "DeepSeek-R1 model card," 2025. [Online]. Available: https://huggingface.co/deepseek-ai/DeepSeek-R1

[45] Google DeepMind, "Gemini 2.5 flash model card," 2025. [Online]. Available: https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-2-5-Flash-Model-Card.pdf

[46] ——, "Gemini 2.5 pro model card," 2025. [Online]. Available: https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-2-5-Pro-Model-Card.pdf

[47] ——, "Gemini 3 flash model card," 2025. [Online]. Available: https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Flash-Model-Card.pdf

[48] ——, "Gemini 3.1 pro model card," 2026. [Online]. Available: https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-1-Pro-Model-Card.pdf

[49] Anthropic, "Claude sonnet 4.6 system card," 2026. [Online]. Available: https://www-cdn.anthropic.com/bbd8ef16d70b7a1665f14f306ee88b53f686aa75.pdf

[50] Confident AI, "DeepEval: The open-source llm evaluation framework," 2023. [Online]. Available: https://github.com/confident-ai/deepeval

[51] Embench, "EMBench-IoT: Embedded benchmark suite for IoT devices," FOSSi Foundation, 2019. [Online]. Available: https://github.com/embench/embench-iot

[52] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)," FIPS PUB 197, 2023. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf