2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 14-15, 2025

# AI Pair or Despair Programming

## Using Aider to build a VIP with UVM-SV and PyUVM

André Winkelmann, Verilab, Munich, Germany (*andre.winkelmann@verilab.com*)
Damir Ahmetovic Ignjic, Verilab, Edinburgh, UK (*damir.ahmetovic@verilab.com*)

***Abstract*—This paper investigates the feasibility of leveraging generative AI coding assistants, specifically Aider, for functional design verification (DV) tasks. By comparing the generation of an AMBA APB3 Verification IP (VIP) in both SystemVerilog with the Universal Verification Methodology (UVM) and Python with cocotb+PyUVM, we assess how mature these AI tools are, in order to support the DV engineer in their day to day coding tasks. The results showcase the strengths and current limitations of generative AI in producing functional and syntactically correct testbenches.**

***Keywords—Aider, generative AI, UVM-SV, PyUVM, LLM, APB, coding assistant***

### I. MOTIVATION

The rapid evolution of generative AI is reshaping industries, with AI-driven coding assistants becoming an integral part of software development. Tools like GitHub Copilot, ChatGPT, Claude Code, Aider [4], Cursor, to name a few, are now widely accepted as standard companions for coding tasks, drastically improving productivity. In the functional DV domain, however, the adoption of such tools offers room for improvement. At first glance, one might expect languages like Python, along with frameworks such as cocotb [7] and PyUVM [8], to align naturally with the strengths of modern AI models, given Python's dominance in software development and the abundance of related training data. Conversely, SystemVerilog (SV) [9] combined with UVM [10], the industry's standard for hardware verification, is more niche, with a steeper learning curve and a perceived lack of accessible, machine-readable training material. This disparity raises critical questions:

- How effective is generative AI at producing cocotb+PyUVM code compared to UVM-SV?
- Is generative AI mature enough to assist DV engineers in creating complex VIP, or is its utility limited to simpler code editing tasks?

This case study seeks to answer these questions, providing valuable insights for DV practitioners evaluating AI's role in their coding workflows. Readers will gain insights into the practical application of generative AI in functional design verification, including detailed comparisons between UVM-SV and cocotb+PyUVM code generation.

### II. METHODOLOGY

To explore these questions, we selected Aider, an AI-powered coding assistant designed for code editing and generation. Aider serves as an interface between human developers and Large Language Models (LLMs), allowing for interactive, guided code development. While many AI discussions focus on RTL generation [3], formal property generation [2] or stimuli generation [1], our focus remains strictly on verification code generation - specifically, generating VIP and self-checking testbenches of the same.

The methodology of this study was designed to systematically evaluate how effectively generative AI, via the Aider coding assistant and leading large language models (LLMs), can produce production-quality Verification IP (VIP) and self-checking testbenches in two verification ecosystems: SystemVerilog with UVM and Python with cocotb + PyUVM. We describe below the tools, configurations, workflows, and evaluation criteria used.

### A. *Aider and its Modes of Operation*

According to its documentation, Aider offers "AI pair programming in your terminal". Under the hood, Aider constructs structured system prompts, passing them to the LLM and applying the responses as concrete code edits or file creations. Aider supports multiple modes of interaction.

- **Terminal mode**: the main interface, using a command-line prompt to communicate with the LLM and edit code.
- **Web interface**: an alternative browser-based UI.
- **In-editor mode**: embedding AI commands directly into code comments, which Aider then executes in-place.

For the purpose of this study, we focused primarily on terminal mode, which allowed us to fully script the workflow and systematically test all model and configuration permutations. The ability to automate Aider via scripts was particularly advantageous in ensuring repeatability and fairness across experiments.

### B. *Experimental Dimensions*

**LLM provider**

The specific LLM models we tested were the top-ranked offerings from each provider at the time of experimentation (as listed on the Aider leaderboards [6]):

- Google: gemini-2.5-pro-preview-06-05
- Anthropic: claude-opus-4-20250514
- OpenAI: o3

**Verification methodology**

- SystemVerilog with UVM
- Python with cocotb + PyUVM

**Coding conventions**

- Without additional coding conventions
- With additional coding conventions [5]

### C. *Project Setup*

For each test run, we prepared a clean directory structure, containing the following components:

```
dut/dut.sv
tb/
tb/tb_apb.f          (UVM-SV only)
tests/
vips/apb/
vips/apb/vlab_apb.f  (UVM-SV only)
Makefile
[conventions.md]     (optional, if conventions were applied)
```

Makefile, dut.sv, and the *.f (file list) files were handcrafted and identical across runs. All other directories and files were generated by Aider during the experiment.

### D. *Prompts and Commands*

To ensure comparability, the same sequence of prompts and commands was used for all LLMs and configurations. Coding conventions (when used) were supplied via Aider's configuration file (.aider.conf.yaml), which pointed to a conventions.md document.

*1)* UVM-SV Flow

**VIP Generation**

aider --yes-always -m "Create under vips/apb directory a production-quality UVM-1.2 SystemVerilog VIP for the AMBA APB3 protocol. The VIP should include all standard UVM components plus a comprehensive sequence library, a functional coverage model and protocol checks implementing the entire APB3 specification."

**Compilation & Fixes**

aider --test-cmd "make compile_vip" --read vips/apb

**Testbench Generation**

aider --read vips/apb --read dut --yes-always -m "Create under tb directory a production-quality self-checking UVM-1.2 SystemVerilog TB for the DUT found under dut/dut.sv which is a pass through of an APB3 master port to an APB3 slave port. The TB should instantiate and hook up the previously created APB3 VIP to the DUT, instantiate a scoreboard to check the master-slave connection. The TB does not need to create a functional coverage model. Under directory tests create a first single test to perform a simple read-write sequence."

**Simulation & Fixes**

aider --test-cmd "make sim" --read vips/apb --read dut --read tb --read tests

*2)* PyUVM Flow

**VIP Generation**

aider --yes-always -m "Create under vips/apb directory a production-quality PyUVM VIP for the AMBA APB3 protocol. The VIP should include all standard UVM components plus a comprehensive sequence library, a functional coverage model and protocol checks implementing the entire APB3 specification."

**Static Analysis & Fixes**

aider --test-cmd "make compile_vip"

**Testbench Generation**

aider --read dut --yes-always -m "Create under tb directory a production-quality self-checking PyUVM TB for the DUT found under dut/dut.sv which is a pass through of an APB3 master port to an APB3 slave port. The TB should instantiate and hook up the previously created APB3 VIP to the DUT, instantiate a scoreboard to check the master-slave connection. The TB does not need to create a functional coverage model. Under directory tests create a first single test to perform a simple read-write sequence."

**Simulation & Fixes**

aider --test-cmd "make sim" --read dut

### *E. Coding Conventions*

We tested each flow both with and without coding conventions. For UVM-SV, we adapted general coding guidelines from Verilab into a conventions.md file in markdown format. For PyUVM, we created a simplified version of the UVM-SV guidelines and iteratively enhanced it based on observed deficiencies during runs without conventions. These conventions were automatically loaded into Aider and provided to the LLM as additional context during code generation.

### *F. Evaluation Procedure*

After generating the VIP and testbench, we analyzed the outputs across several dimensions.

- **Review of the generated VIP code**: Compared code completeness and code quality between LLMs and between conventions and no conventions being used. That step was only performed for the UVM-SV runs.
- **Syntactic correctness & elaboration**
  - For UVM-SV: used commercial simulator to compile and elaborate the code.
  - For PyUVM: used mypy as a static code analyzer.
  - This was only run for permutations without coding conventions
- **Iteration count VIP**: Tracked how many Aider iterations were needed (with or without additional human prompting) to achieve compiling and elaborating code for the VIP alone. This was only run for permutations without coding conventions
- **Iteration count TB+VIP**: Similarly, counted iterations required to generate a functioning testbench with the VIP integrated and running to completion in simulation. This was only run for cases without coding conventions.
- **Functional correctness**: Once the testbench ran, we inspected the initial simulation waveforms to verify the correctness of APB3 transfers. This was run for cases without coding conventions. For PyUVM, note that in the final step we focused on running the simulation directly without further mypy checks at that point.
- **LLM Costs**: Aider prints out the costs of each LLM request. That is summed up and reported for generation of VIP and for overall TB+VIP debug.

## III. RESULTS

### *A. UVM-SV VIP code review*

The first step was to look at the generated VIP code and analyse if additional coding conventions help the LLM to produce more industry aligned code. Overall it is surprising how different the code generation results are between the LLMs.

Google's LLM benefits notably from being guided. While it doesn't reach the completeness or depth of Anthropic, conventions help it build a more structured and usable baseline.

Anthropic is capable of rich and technically aware output without guidance, but its response to conventions may lead to over-conformity at the cost of functionality. It excels when allowed to "think freely."

OpenAI's LLM responds well to structured prompting, producing technically aligned output with more formalism. However, gaps in configurability and verification depth remain. It trades breadth for cleanliness.

The following high-level trends emerged:

- **Conventions Improve Structural Quality**: All LLMs showed some form of improvement in file organization, naming, or modularity when prompted with conventions. File boundaries were respected more consistently, and reusable types were often abstracted properly.
- **Loss of Functionality Is a Risk**: While conventions guide structure, they sometimes lead to oversimplification. Anthropic, in particular, removed useful protocol logic (reset handling, pready delay monitoring) under convention pressure — highlighting the trade-off between conformity and insight.
- **Protocol Versions Are a Common Pitfall**: Despite clear APB3 context, APB4 signals (like pstrb, pprot) were frequently included across all vendors — both with and without conventions — indicating a misunderstanding of specification scoping or a lack of contextual filtering.

- **Coverage Quality Varies Greatly**: Anthropic and OpenAI both managed to produce meaningful coverage bins and crosses (especially under guidance). Google, by contrast, lagged behind, often skipping coverage entirely or defining very basic logic.
- **Reset and Timing Are Weak Points**: Few implementations across the board handled reset synchronously and comprehensively. Misuse of clocking blocks and delayed inputs was prevalent, especially with Google and Anthropic.
- **Sequence Libraries Show Promise**: All models attempted to build sequence libraries with write/read support. Anthropic led in terms of breadth (burst, random, walking patterns), while OpenAI and Google were more minimal.

The following section shows some examples of how the code conventions influenced the outcome.

*Convention: Use the covergroup sample() method to collect coverage*

| Without convention | With convention |
|---|---|
| covergroup apb_cg; | covergroup apb_cg with function sample(apb_transaction trans); |

*Convention: Use prefix_ and _postfix to delineate name types*

| Without convention | With convention |
|---|---|
| virtual apb_if vif;<br>apb_config cfg; | virtual apb_if m_vif;<br>apb_config m_config; |

*Convention: Use a begin-end pair to bracket conditional statements*

| Without convention | With convention |
|---|---|
| if (!uvm_config_db#(apb_config)::get(this, "", "cfg", cfg))<br>`uvm_fatal("NOCFG", "No apb_config found") | if (!uvm_config_db#(apb_config)::get(this, "", "cfg", m_cfg)) begin<br>`uvm_fatal("NOCFG", "No apb_config found")<br>end |

## B. *Syntactic correctness of VIP code*

This was one of the huge surprises that none of the LLMs, regardless of being asked to generate PyUVM or SV-UVM, generated compile and elaboration clean code. Table I shows the iteration count that was needed to achieve error free code. It distinguishes between full automatic code changes and additional user prompting being necessary. E.g. if the table shows 3(4) it means 3 out of 4 issues could be solved automatically.

Table I.

| Generated Framework | LLM | | |
|---|---|---|---|
| | ***Anthropic Opus*** | ***Google Gemini*** | ***OpenAI o3*** |
| UVM-SV | 1(1) | 1(3) | 0(3) |
| PyUVM | 1(1) | 0(3) | 0(3) |

The authors noted the following general observations while fixing compile time issues with the help of Aider and the different LLMs:

- Multiple automatic bug fix runs from the same starting point produce different results
  - Seems some heuristic gets applied
  - Sometimes able to fix the code, sometimes not
  - Thinking sometimes suggests two potential fixes and it seems to roll a dice to pick one of those or let the user choose
- All LLMs work like trial and error. Especially with the "thinking/reasoning" enabled one can read the step by step reasoning like a human engineer would do: Trial and error. The LLMs literally write into their reasoning output "Let's try xyz and see if that fixes the issue". This is no surprise, given the LLMs are trained with what we humans wrote about debugging.
- The LLMs don't seem to understand PyUVM at a similar level like other more widely used Python standard libraries. The authors got the impression that the LLMs mostly try to guess that whatever worked in UVM-SV might work as well in PyUVM. The differences though do not seem to be understood well as the more detailed results show in the later chapters.

### C. *LLM iterations needed for first simulation*

All three LLMs performed at a similar level when it came to compiling and running the entire UVM-SV TB+VIP. On the PyUVM side though the results were not as encouraging. The authors started with the Google Gemini LLM but in the end spent 4 hours running through 33 iterations of mostly automatic bug fixing. Anthropic we had to abort and OpenAI was very slow to respond. Table II shows the iteration count that was needed to achieve a complete simulation run. It distinguishes between full automatic code changes and additional user prompting being necessary. E.g. if the table shows 3(4) it means 3 out of 4 issues could be solved automatically.

Table II.

| **Generated Framework** | **LLM** | | |
|---|---|---|---|
| | ***Anthropic Opus*** | ***Google Gemini*** | ***OpenAI o3*** |
| UVM-SV | 4(5) | 3(3) | 5(6) |
| PyUVM | 20(>24 aborted) | 25(33) | 16(19) |

### D. *Waveform analysis*

- PyUVM Google Gemini
  - Despite the prompt asking for a simple read write test, the waves show only read transactions and all to the same address
  - No signals are X or Z
  - The transactions are two cycle transactions but the slave asserts PREADY only after seeing both PSEL and PENABLE, which turns the transactions into 3 cycle long
- PyUVM Anthropic Opus
  - No waves analyzed as fixing bugs took too many iterations
- PyUVM OpenAI o3
  - All signals HiZ as the code misses the DUT to VIP hookup code
- UVM-SV Google Gemini

- It correctly shows a write and a read transaction, both 2 cycles long back to back
- Transactions happen though while still in reset
- The second APB3 agent configured as passive, does not drive PREADY and PSLVERR, hence those show as X

- UVM-SV Anthropic Opus
  - Well formed read and write transactions, each 2 cycles long
  - One idle cycle between transactions though
  - Memory model in TB attached to slave port
  - No signals X
  - Waveform shows what got written into memory gets read correctly
- UVM-SV OpenAI o3
  - apb_driver is not driving nor waiting on the PRESETn signal to be deasserted before starting the transfers
  - apb_driver implementing 3+ cycle transfers
  - PREADY is driven by another module generated by the LLM to represent a simple memory slave
  - No signals are X

### *E. Cost analysis*

Table III reports the costs of the first VIP generation. Table IV reports the overall cost of VIP+TB generation plus bug fixing iterations.

Table III. VIP generation cost in USD

| **Generated Framework** | **LLM** | | |
|---|---|---|---|
| | ***Anthropic Opus*** | ***Google Gemini*** | ***OpenAI o3*** |
| UVM-SV | 0.88 | 0.08 | 0.10 |
| UVM-SV conv | 0.84 | 0.13 | 0.09 |
| PyUVM | 0.92 | 0.08 | 0.06 |
| PyUVM conv | 0.85 | 0.10 | 0.11 |

Table IV. Overall cost in USD

| **Generated Framework** | **LLM** | | |
|---|---|---|---|
| | ***Anthropic Opus*** | ***Google Gemini*** | ***OpenAI o3*** |
| UVM-SV | 8.35 | 0.62 | 0.98 |
| PyUVM | 29.94 | 6.76 | 1.06 |

## IV. Conclusion

This study set out to explore two key questions.

1. How effective is generative AI at producing cocotb+PyUVM code compared to UVM-SystemVerilog?
2. Is generative AI mature enough to assist DV engineers in creating complex VIP, or is its utility limited to simpler code editing tasks?

Based on our experiments, we conclude that, as of today, generating UVM-SV code appears more mature and reliable than generating cocotb+PyUVM code. Despite the initial expectation that Python-based frameworks might align better with AI models due to Python's prevalence and the good documentation of cocotb and PyUVM, the actual results suggest otherwise. One likely explanation is that UVM-SV enjoys a richer ecosystem of publicly available examples, tutorials, and community content, which likely provided the LLMs with more relevant training material. In contrast, cocotb and PyUVM, while well-documented, have far fewer publicly available code examples, limiting the LLMs' familiarity with them.

Another key insight is that achieving high-quality outputs — particularly for complex VIP — requires carefully crafted, detailed prompts. Including specific details such as master/slave roles, active/passive agent behavior, and explicit coverage or protocol-checking requirements consistently improved the generated results. Vague or minimal prompts tended to produce incomplete or superficial solutions. Similarly, providing coding conventions as additional context helped steer the LLMs toward more structured and maintainable code. However, we observed that strict conventions sometimes constrained the models in ways that reduced functional completeness, suggesting a need to develop conventions specifically tailored for LLM consumption rather than reusing traditional human-oriented guidelines.

Importantly, during the debugging and correction phase, we observed that LLMs were often able to propose plausible solutions to compilation or simulation errors far faster than a human engineer would have identified them — though this was not always the case. While their suggestions sometimes required further refinement, the ability to quickly generate hypotheses and corrective code proved valuable. This suggests that generative AI is not just a code generator but also a potentially powerful partner in diagnosing and resolving issues, especially when guided by an experienced engineer who can validate and correct its output efficiently.

Overall, our findings suggest that AI coding assistants are already mature enough to contribute meaningfully to verification workflows, particularly when generating block-level VIP and self-checking testbenches, and when used as a debugging aid. They enable engineers to spend less time on boilerplate and more time architecting, reviewing, and refining. The combination of human expertise and AI assistance appears to yield the best outcomes.

## References

[1] D. N. Gadde, A. Kumar, T. Nalapat, E. Rezunov, F. Cappellini, "All Artificial, Less Intelligence: GenAI through the Lens of Formal Verification", DVCon US 2024

[2] H. A. Quddus, Md S. Hossain, Z. Cevahir, A. Jesser, Md N. Amin, "Enhanced VLSI Assertion Generation: Conforming to High-Level Specifications and Reducing LLM Hallucinations with RAG", DVCon EU 2024

[3] B. Cohen, "Leveraging Bing GPT-4 for Digital Design and Verification with SystemVerilog and Assertions", https://systemverilog.us, 2023

[4] "Aider", https://aider.chat

[5] "Aider: Adding coding conventions", https://aider.chat/docs/usage/conventions.html

[6] "Aider: LLM leaderboards", https://aider.chat/docs/leaderboards

[7] "Cocotb", https://www.cocotb.org

[8] "PyUVM", https://github.com/pyuvm/pyuvm

[9] "IEEE Std 1800™-2023 IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language", IEEE Computer Society

[10] "IEEE Std 1800.2™-2020 - IEEE Standard for Universal Verification Methodology Language Reference Manual", IEEE Computer Society