POLITECNICO DI TORINO
Repository ISTITUZIONALE

Si-ATLAS: Autonomous Testing via LLM-Agent Synergy for RISC-V SBST Generation

Original
Si-ATLAS: Autonomous Testing via LLM-Agent Synergy for RISC-V SBST Generation / Wang, C., Chen, Y., Bellarmino, N., Rodriguez Condia, J.E., Li, C., Squillero, G., Cantoro, R.. - (In corso di stampa). (International Test Conference 2026 (ITC 2026) San Antonio, Texas, USA October 11–16, 2026).

Availability:
This version is available at: 11583/3016158 since: 2026-09-30T11:58:33Z

Publisher:
IEEE

Published
DOI:

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©9999 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

01 November 2026

Si-ATLAS: Autonomous Testing via LLM-Agent Synergy for RISC-V SBST Generation

Changhao Wang*†, Yijing Chen*, Nicolò Bellarmino*, Josie Esteban Rodriguez Condia*
Chaobo Li†, Giovanni Squillero*, Riccardo Cantoro*
*Politecnico di Torino, Torino, Italy
{changhao.wang, nicolo.bellarmino, giovanni.squillero, josie.rodriguez, riccardo.cantoro}@polito.it
s336790@studenti.polito.it
†Institution of Microelectronics, Chinese Academy of Sciences, Beijing, China
lichaobo@ime.ac.cn

***Abstract*—Software-Based Self-Test (SBST) generation remains expert-intensive because useful programs must activate internal faults, propagate their effects, and make them architecturally observable. We present Si-ATLAS, a closed-loop framework that converts gate-level fault-simulation results into generation constraints. It combines deterministic instruction templates, LLM-guided structural exploration, and AST-aware operand fuzzing, while a static sanitizer and a runtime watchdog isolate invalid or non-terminating programs. On the CV32E40P RISC-V core, Si-ATLAS reaches 83.26% full-core stuck-at fault coverage; the multiplier reaches 95.38% after seven iterations. The default Claude Sonnet 3.5 campaign costs $5.23, while DeepSeek-V3 reduces the API cost to $0.64 and retains competitive coverage on regular datapaths.**

I. Introduction

Software-Based Self-Test (SBST) exposes structural processor faults by executing specialized instruction sequences and observing their effects in registers, memory, or control flow. A useful sequence must sensitize a fault, propagate the resulting error through the pipeline, and capture it at an architecturally visible state [1, 2]. Consequently, legal assembly is not sufficient: instruction ordering, operands, hazards, stalls, and observation operations must jointly create the required microarchitectural state.

Existing automation faces a scalability–portability trade-off. Pseudo-random generation scales but poorly targets deep control paths. Formal and constraint-based methods provide structural direction, yet sequential unrolling becomes expensive for complete pipelined cores; published studies therefore often use simplified cores or selected modules [3, 4]. Expert-written Software Test Libraries (STLs) avoid that state-space explosion, but their construction and migration to new cores remain costly [5].

Large Language Models (LLMs) can generate ISA-level instruction patterns without explicitly unrolling the gate-level state space and have shown value in hardware code generation [6, 7]. Used alone, however, they lack physical feedback and may emit unsupported instructions, illegal addresses, or unsafe control flow. They also cannot determine whether an instruction sequence actually detects a physical fault.

Si-ATLAS addresses this gap with a bounded, feedback-driven search. Gate-level fault classes are converted into explicit generation actions, and three complementary engines divide structural, semantic, and combinatorial work. The contributions of this short paper are: (1) a closed-loop formulation that uses fault-simulation data as an active control signal; (2) a multi-engine generator with static and runtime guardrails; and (3) a full-core evaluation on the four-stage RV32IMC CV32E40P showing 83.26% coverage at low API cost.

II. Background and Motivation

A. Structural Requirements of Processor SBST

SBST differs from functional validation because correct program execution does not imply that the underlying hardware has been structurally exercised. A workload may cover many ISA behaviors while leaving forwarding multiplexers, hazard logic, pipeline flush paths, or arithmetic corner cases inactive. Test quality must therefore be measured against a structural fault model rather than instruction or line coverage alone [8, 9].

The connection between a gate-level fault and a software-visible deviation is indirect. For example, a fault in decode or forwarding logic may require a specific producer–consumer distance, operand relation, and subsequent store operation. Branch and exception logic additionally depends on transient pipeline states that cannot be addressed directly from software. Effective generation is consequently a search over instruction topology, operand data, and observation paths, not merely a search for syntactically legal programs [10, 11].

B. Why a Closed Loop is Needed

Formal engines can derive structurally targeted patterns by translating fault conditions into Boolean constraints. Their precision is valuable, but memory and runtime grow rapidly when a complete pipeline, memory latency, branch recovery, and multiple clock cycles are unrolled. Manual STLs avoid explicit unrolling through expert knowledge, yet that knowledge is embedded in architecture-specific routines and must be revised when the ISA, memory map, or microarchitecture changes.

An LLM offers a complementary abstraction: it can propose macro-level instruction dependencies at the ISA level. Nevertheless, an open-loop prompt cannot establish whether those dependencies toggled a target net or propagated a fault.

Algorithm 1 Feedback-Driven Batch Workflow

Require: target C_t, batches B, iterations K

$\mathcal{S} \leftarrow \emptyset$; $C_{best} \leftarrow 0$
for $b = 1$ to B **do**
  if $C_{best} \geq C_t$ **then**
    break
  end if
  $Plan \leftarrow \textsc{Orchestrator}(\mathcal{S})$
  $R \leftarrow \emptyset$
  for $k = 1$ to K **do**
    $T \leftarrow \textsc{Generate}(Plan)$
    $(C, r) \leftarrow \textsc{ExecuteAndGrade}(T)$
    $C_{best} \leftarrow \max(C_{best}, C)$; $R \leftarrow R \cup \{r\}$
    if $\textsc{Plateau}(R)$ **then**
      break
    end if
  end for
  $\mathcal{S} \leftarrow \textsc{CompressState}(R)$
end for

It may repeatedly produce plausible but structurally redundant code. Si-ATLAS therefore treats the LLM as one search engine inside a physical feedback loop. Fault grading supplies the objective signal; deterministic validation and local fuzzing constrain the LLM's non-determinism. This design preserves exploratory capability without assigning physical correctness to the language model itself.

III. Feedback-Driven Generation Framework

Fig. 1 shows the four-stage workflow. An outer batch loop plans from aggregated physical feedback, while an inner loop generates and evaluates up to five programs under a fixed plan. We use at most four batches (20 iterations). A batch terminates early when consecutive iterations improve coverage by less than 0.1%. This hierarchy limits API use while allowing the planner to change strategy after a local plateau.

Algorithm 1 summarizes the bounded control flow. The Orchestrator is called only at batch boundaries, so its plan remains fixed during local exploration. Each inner iteration generates one candidate, executes the backend flow, and records both coverage and failure status. At a plateau, the batch results are compressed into the next state vector, forcing the subsequent plan to reconsider the engine choice or structural target.

A. Fault-Driven Orchestration

The Orchestrator receives a compressed state vector containing uncovered hierarchical signal paths, fault classes, recent coverage gains per engine, and hashes of unproductive instruction skeletons. A ReAct-style query–analyze–decide–schedule loop [12] maps this state to a batch plan specifying the target module, instruction class, operand seeds, and selected generation engines. A knowledge base restricts the plan to the RV32IMC ISA, implemented CSRs, exception behavior, and valid memory ranges. Rule-based validation rejects unsupported extensions or configurations before they reach code generation.

The state vector is intentionally smaller than the raw simulation report. Fault-simulation logs can contain thousands of hierarchical paths and repeated status messages, which would waste context and obscure the most actionable signals. The parser therefore retains the top uncovered paths per fault class, their architectural hierarchy, the coverage delta of the engine that produced the program, and a short history of failed strategies. Previously generated instruction skeletons are normalized and hashed so that operand-only changes do not disguise a repeated topology. The Orchestrator sees this compact record rather than complete EVCD traces or netlists.

The validated plan separates goals from executable code. It records a target module, focus instructions, desired dependency patterns, seed operands, and a budget for each synthesis engine. Keeping this representation machine-readable allows deterministic checks before an LLM response is accepted. For example, a plan requesting a floating-point operation on RV32IMC, an unavailable CSR, or an address outside the configured memory map is returned to the planning loop with an explicit error. The resulting correction is cheaper and safer than discovering the same violation after compilation or simulation.

The feedback mapping is deliberately operational. *Not Controlled* faults indicate insufficient activation and therefore request new instruction paths from the exploratory engine. *Not Detected* or *Not Observed* faults have been activated but not propagated; they request operand mutation or explicit observation sequences. Detected faults update the engine-history score, biasing later plans toward strategies that produced measurable gains.

B. Multi-Engine Synthesis and Guardrails

The deterministic engine establishes a valid prefix with regular structures such as March-like memory accesses and read-after-write hazard templates [11, 13]. For control-dependent targets, the exploratory LLM emits an instruction skeleton containing typed operand slots rather than exhaustively enumerating constants. An AST-aware micro-fuzzer fills these slots from boundary values, alternating bit patterns, legal registers, and aligned addresses. If a skeleton has pools P_k, its candidate space is $|V| = \prod_k |P_k|$; sampling caps each skeleton at 500 variants. This division leaves topology discovery to the LLM and moves large, zero-cost operand sweeps to a local deterministic tool [14, 15].

A Sanitizer rejects writes to protected regions, invalid opcodes, misaligned control transfers, and malformed runtime code. Accepted tests are compiled, simulated on the gate-level netlist, and fault graded. A hardware Watchdog terminates programs whose trace exceeds 50 MB or whose runtime exceeds a dynamic instruction-count-based limit. Thus, unstable model output cannot indefinitely block the EDA flow; failed iterations return a terminal status and no coverage gain.

Backend results are normalized before the next planning step. Besides overall coverage, the parser records detected, not-controlled, not-detected, not-observed, non-testable, and possibly-detected populations for each module. It also records

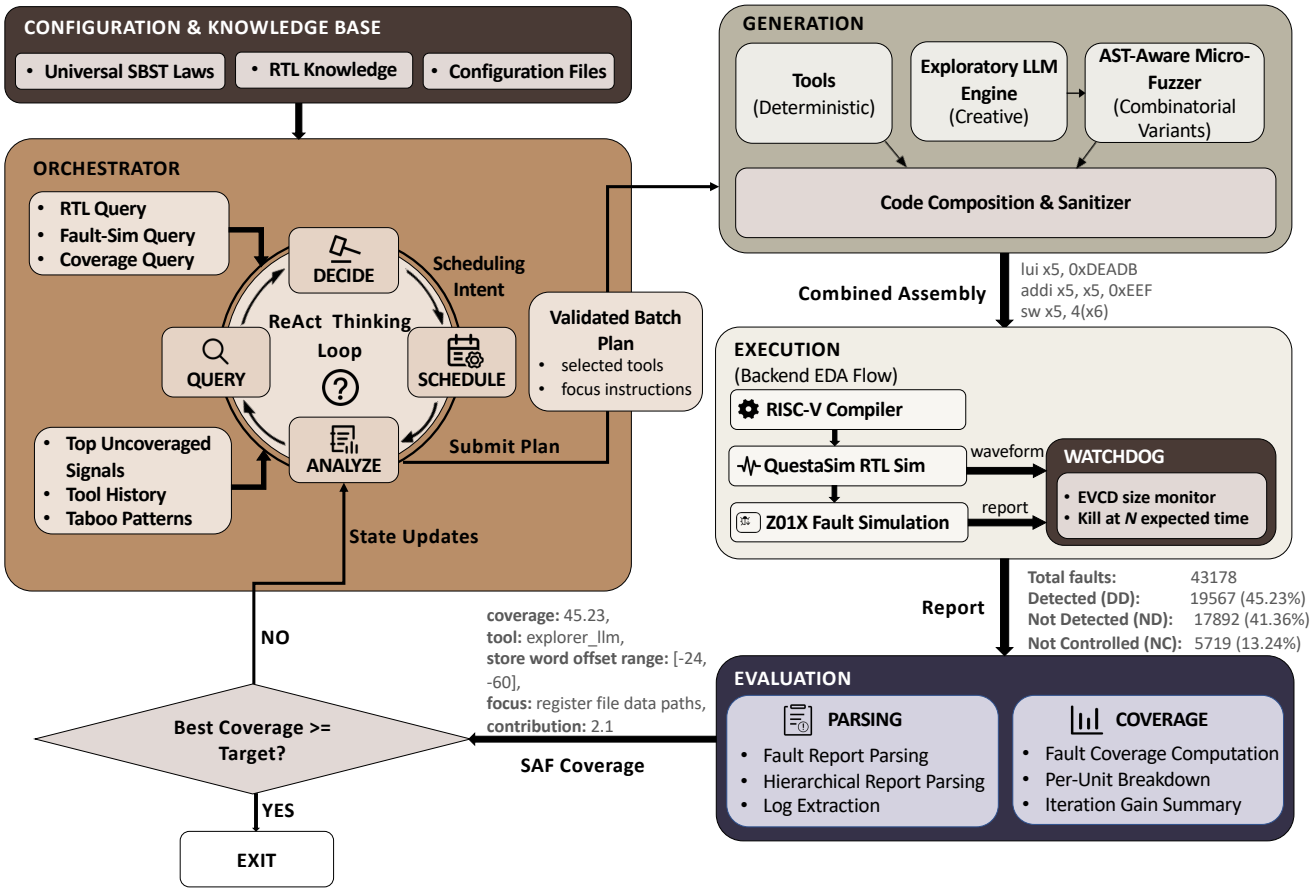


Fig. 1. Si-ATLAS closed loop. The Orchestrator turns compressed fault feedback into a validated batch plan. Deterministic templates, an exploratory LLM, and an AST-aware micro-fuzzer generate candidates; the Sanitizer and Watchdog bound their use in the backend toolchain.

compiler rejection, simulation timeout, trace overflow, and zero-gain outcomes. These statuses distinguish a structurally unproductive program from a program that never reached fault simulation. Coverage is kept monotonic by retaining the best valid test set, while unsuccessful candidates still update the taboo history and safety statistics.

The plateau rule is evaluated locally within a batch rather than globally. A small delta does not terminate the campaign; it ends the current plan and returns control to the Orchestrator with newly aggregated evidence. Thus, a productive operand sweep may continue for several iterations, whereas repeated zero-gain skeletons quickly trigger replanning. Global termination occurs only when the target is reached or the four-batch budget is exhausted.

The three engines are selected independently rather than executed unconditionally. Regular arrays and datapaths initially favor deterministic templates. A high NC population increases the priority of the exploratory LLM because new paths or instruction dependencies are required. A high ND/NO population instead favors operand mutation and appended observation code. This selection policy prevents expensive LLM calls from being spent on exhaustive constant enumeration and makes the coverage delta attributable to a concrete generation action.

IV. Experimental Evaluation

A. Setup

We evaluate the open-source CV32E40P, a four-stage, in-order RV32IMC core [16, 17]. RISC-V assembly is compiled with GCC and simulated on the synthesized gate-level netlist using Siemens QuestaSim. EVCD activity traces are graded with Synopsys Z01X under the stuck-at fault model. All LLM backends use the same 20-iteration limit, Sanitizer, Watchdog, and plateau rule. Claude Sonnet 3.5 is the default backend.

The full-core report uses eight disjoint architectural partitions whose fault counts sum to 130,374. Some convergence and backend analyses use finer or grouped functional units to expose model sensitivity; these analytical groups must not be summed as a second full-core total. Each backend starts from the same template-only baseline and receives the same batch and iteration budgets. This controls search depth, although model-specific tokenization and response length naturally lead to different token counts and monetary costs.

B. Coverage and Convergence

Table I reports 108,549 detected faults out of 130,374, for 83.26% full-core coverage. Regular datapaths are the most software-testable: the multiplier reaches 95.38%, the register

TABLE I
FULL-CORE SAF RESULTS WITH CLAUDE SONNET 3.5. ROWS ARE DISJOINT PARTITIONS AND SUM TO THE CORE TOTAL.

Architectural Module	Total	Detected	ND	NC	NO	NT	PD	SAF Coverage
Instruction Fetch Stage	13,752	8,957	552	1,839	2,347	23	34	65.13%
Compressed Decoder	3,674	3,119	130	118	86	221	0	84.89%
Register File	43,178	39,849	1,071	96	257	1,905	0	92.29%
Execute Residual Logic	17,930	15,314	72	197	665	1,682	0	85.41%
Arithmetic Logic Unit	5,916	5,438	461	4	10	0	3	91.92%
Multiplier	15,744	15,017	361	0	53	312	1	95.38%
Load/Store Unit	5,912	5,152	47	15	526	172	0	87.14%
Controller	24,268	15,703	796	321	4,017	3,425	6	64.71%
Overall Core	**130,374**	**108,549**	**3,490**	**2,590**	**7,961**	**7,740**	**44**	**83.26%**

ND: Not Detected; NC: Not Controlled; NO: Not Observed; NT: Non-Testable; PD: Possibly Detected.

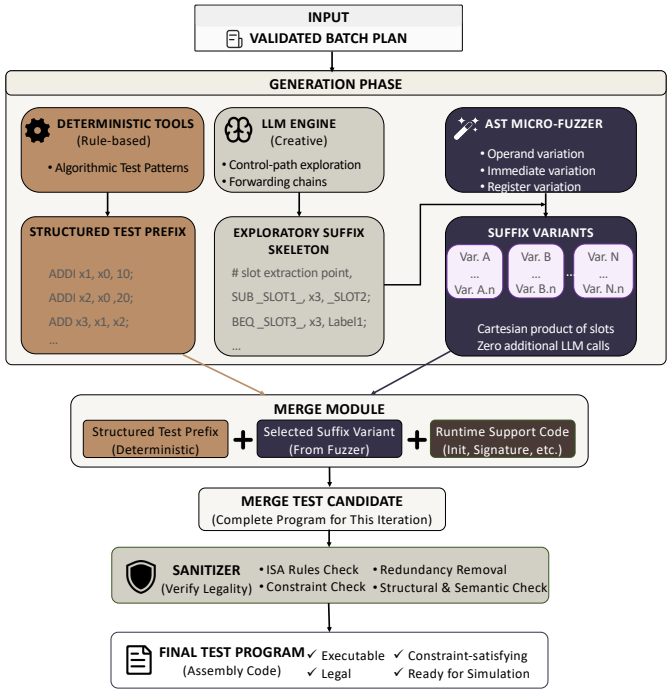


Fig. 2. Multi-engine synthesis: a deterministic prefix establishes legal state, the LLM proposes a structural skeleton, and the micro-fuzzer expands typed operand slots before sanitization.

file 92.29%, and the ALU 91.92%. Load/store logic reaches 87.14%. In contrast, instruction fetch and controller logic reach 65.13% and 64.71%, respectively. Their residual faults are dominated by unobserved or non-testable sites tied to global pipeline control, exceptions, and states that software cannot directly expose.

Fig. 3 demonstrates why feedback is useful beyond final grading. The multiplier exceeds 95% by iteration 7 because a valid instruction topology can be expanded efficiently through operand sweeps. ALU and decode coverage rises more gradually as new operation and control contexts are introduced. Control-state logic plateaus lower, exposing an architectural observability limit rather than a syntax-generation limit.

C. Comparison and Backend Cost

Table II places the result in context. The automated formal baselines report smaller evaluation spaces, whereas Si-ATLAS

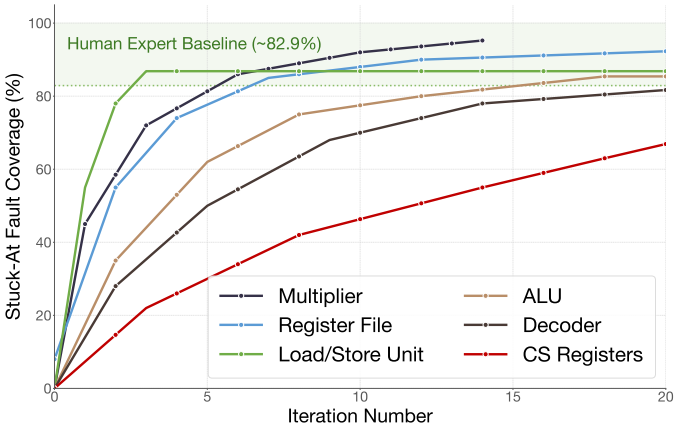


Fig. 3. Coverage across generation iterations. Regular datapaths improve rapidly, whereas control/state-dependent modules require sustained search.

grades a complete four-stage core. The closest full-core reference is the manually constructed RI5CY STL: Si-ATLAS obtains comparable coverage with a test image that is 18.8% smaller. Differences in cores, fault lists, and grading flows prevent a strict head-to-head ranking, but the comparison shows that feedback-guided generation scales to a fault population comparable to a manual full-core study.

Table III shows that the framework is not tied to one model. Sonnet 3.5 provides the most consistent result across datapath and control modules. DeepSeek-V3 is strongest on the multiplier and ALU/divider and costs only $0.64, but plateaus earlier on control-heavy modules. Gemini is strongest on compressed decode. GPT-4o frequently triggers guardrail termination; its small token bill therefore reflects early stopping rather than higher search efficiency. Across all backends, the Sanitizer and Watchdog keep invalid generations from destabilizing compilation and simulation.

D. Resource Allocation Across Modules

Fig. 4 separates the API expenditure of the Orchestrator from that of the Explorer. The total Sonnet 3.5 campaign costs $5.23, and no individual module exceeds $1.60. The multiplier and register file require only $0.22 and $0.30, respectively, because a small number of legal skeletons can be expanded locally over many operand combinations. This is the intended economic effect of the multi-engine design: the LLM identifies

TABLE II
COMPARISON WITH REPRESENTATIVE SBST APPROACHES.

Work	Method	Core	ISA	Stuck-at faults	Code size	SAF Coverage
Faller et al. [3]	Automated BMC+VCM	DarkRISCV	RV32I, 3-stage	29,071	26.1 KB	79.18%
Kaja et al. [18]	Automated formal+ATPG	Proprietary	RV32IMC	∼20,224	N/R	∼81.74%
Cantoro et al. [19]	Manual expert STLs	RI5CY	RV32IMC, 4-stage	141,990	88.4 KB	82.90%
Si-ATLAS	**Automated LLM-guided**	**CV32E40P**	**RV32IMC, 4-stage**	**130,374**	**71.8 KB**	**83.26%**

TABLE III
BACKEND SENSITIVITY, TOKEN USE, AND API EXPENDITURE.

Module/Metric	Template Only	Sonnet 3.5	DeepSeek-V3	Gemini 2.5 Pro	GPT-4o
Multiplier	51.00%	95.38% (E7)	**96.04%** (E3)	95.48% (E3)	89.67% (K15)
ALU & Divider	45.00%	91.92% (E20)	**92.05%** (P10)	91.14% (P12)	85.33% (K14)
Compressed Decoder	48.00%	84.89% (R)	88.42% (P14)	**89.15%** (P14)	74.21% (K12)
Load/Store Unit	30.00%	**87.14%** (E3)	71.35% (P15)	75.82% (P17)	58.46% (K10)
Register File	0.00%	**92.29%** (E8)	81.34% (P20)	57.71% (R)	0.00% (K5)
Controller	0.00%	**64.71%** (R)	55.18% (P13)	52.71% (P16)	38.71% (K19)
Input / output tokens	–	742K / 200K	1.49M / 217K	1.22M / 172K	0.31M / 16K
Total cost	–	$5.23	**$0.64**	$3.24	$0.93

E: exceeds the manual reference at iteration n; P: plateaus at n; R: still rising at the iteration limit; K: stopped by guardrails at n.

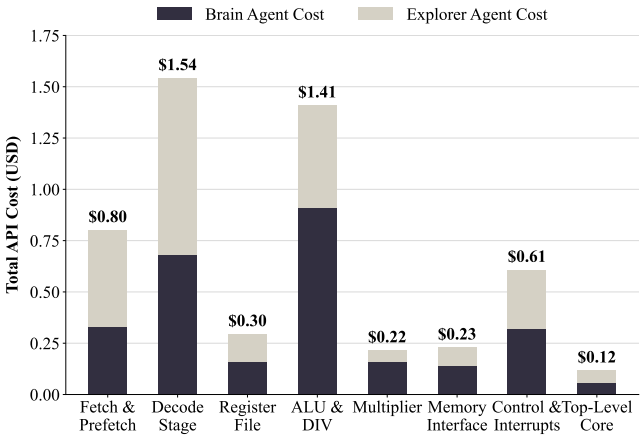


Fig. 4. Cumulative API cost by architectural module and agent role. The Orchestrator performs planning, while the Explorer generates structural suffixes; deterministic fuzzing incurs no API cost.

useful structure, while repeated arithmetic and data-pattern exploration is handled without further API calls.

Instruction decode and ALU/divider logic consume the largest budgets ($1.54 and $1.41). Decode requires diverse legal instruction contexts and therefore draws more Explorer tokens. ALU/divider and controller progress depends more on choosing corner cases, CSR configurations, and dependent sequences, which increases the Orchestrator share. Cost thus follows the difficulty of finding new structural contexts rather than final coverage alone. A module can reach high coverage cheaply when its remaining search space is regular, while a lower-coverage control block may consume more planning effort before reaching an observability limit.

V. DISCUSSION AND LIMITATIONS

A. Interpreting Residual Fault Classes

The final fault distribution provides diagnostic information beyond the 83.26% aggregate. The multiplier has no NC faults and only 53 NO faults, indicating that software can activate nearly all of its regular datapath and that the remaining gap is small. The controller, by contrast, contains 4,017 NO and 3,425 non-testable faults. Instruction fetch contains 1,839 NC and 2,347 NO faults. These populations explain why additional operand mutation continues to help arithmetic modules but yields diminishing returns on global control logic. For the latter, architectural visibility and hardware-defined states dominate the remaining gap; generating more legal assembly is not by itself sufficient.

This distinction also prevents the feedback loop from treating every plateau identically. A plateau dominated by NC faults motivates new control-flow or instruction-class exploration. A plateau dominated by NO faults motivates stores, register transfers, or exception handlers that improve propagation. A large non-testable population instead signals that the current software-only interface has reached a structural boundary and that design-for-test support or a different observation mechanism may be necessary.

B. Scope and Threats to Validity

The current evaluation targets one single-issue, in-order processor and one commercial gate-level tool flow. Absolute coverage is affected by synthesis, fault collapsing, library models, and simulator classification rules, which is why cross-paper results in Table II are contextual rather than strictly equivalent. Backend prices can also change over time; the reported costs should be interpreted as the expenditure of the evaluated campaigns, not as a permanent price guarantee.

The compressed state vector represents temporal behavior indirectly through fault classes, signal hierarchy, and recent gains. This is adequate for the four-stage CV32E40P but does not explicitly encode the lifetime of hazards or the ordering of multiple in-flight instructions. Superscalar and out-of-order cores will require cycle-accurate traces of issue queues, reorder buffers, and speculative recovery. A second extension is graph-based RTL context: netlist connectivity and forwarding paths could provide the Orchestrator with explicit hardware proximity instead of requiring it to infer locality from hierarchical signal names. Evaluating these extensions across multiple cores and technology libraries is necessary to establish portability beyond the present case study.

C. Portability and Reproducible Deployment

Porting the framework to another in-order RISC-V core primarily requires changing declarative inputs rather than rewriting the generation loop. The configuration layer identifies the implemented ISA extensions, legal CSR set, exception behavior, linker memory map, boot sequence, and backend invocation commands. The structural layer supplies the synthesized design hierarchy and fault report schema used by the parser. Once these inputs are available, the same Orchestrator interface, typed skeleton representation, Sanitizer, and plateau policy can be reused. Core-specific deterministic templates may still be added for memories or proprietary units, but they remain isolated tools rather than assumptions embedded throughout the control flow.

For backend comparisons, reproducibility also requires separating framework policy from model behavior. In this evaluation, all models share the same configuration, initial templates, maximum of four batches and five iterations per batch, plateau threshold, Sanitizer rules, and Watchdog limits. The reported token counts include both planning and exploratory generation, while local deterministic generation and micro-fuzzing contribute no API tokens. This accounting makes clear whether a low monetary cost comes from efficient progress or from early guardrail termination. It also permits future models to be substituted without changing the physical feedback, execution, or grading criteria used to judge their generated programs.

Batch-level checkpoints further reduce dependence on a particular service or tool session. A campaign can resume from the last compressed state, replay the validated plan with stored seeds, or replace only the exploratory backend. This makes failures auditable and avoids repeating completed gate-level fault simulations when generation is interrupted.

VI. Conclusion

Si-ATLAS turns physical fault-simulation feedback into bounded generation actions for RISC-V SBST. Its deterministic, LLM-guided, and fuzzing engines separate regular operand exploration from difficult instruction-topology search, while guardrails protect the backend flow. On CV32E40P it achieves 83.26% full-core SAF coverage with a 71.8 KB test suite and modest API cost. Future work will add cycle-accurate state descriptions and graph-based RTL context to target timing-sensitive structures in superscalar and out-of-order cores.

Acknowledgment

This work is supported in part by the EU project titled RIGOLETTO. The RIGOLETTO project is funded by the Chips Joint Undertaking (Chips JU) under Grant Agreement No. 101194371.

References

[1] W. Changhao et al., "Defects, fault modeling, and test development framework for fefets," in *International Test Conference (ITC) 2024*, IEEE, 2024.

[2] A. Paschalis et al., "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *IEEE Transactions on Computer-aided design of integrated circuits and systems*, vol. 24, no. 1, pp. 88–99, 2004.

[3] T. Faller et al., "Constraint-based automatic sbst generation for risc-v processor families," in *2023 IEEE European Test Symposium (ETS)*, 2023, pp. 1–6.

[4] L. Weingarten et al., "Complete and efficient verification for a risc-v processor using formal verification," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2024, pp. 1–6.

[5] P. Bernardi et al., "Fault grading of software-based self-test procedures for dependable automotive applications," in *2011 Design, Automation & Test in Europe*, IEEE, 2011, pp. 1–2.

[6] D. Schwachhofer et al., "Large language model-based optimization for system-level test program generation," in *2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, IEEE, 2024, pp. 1–6.

[7] R. Ma et al., "Verilogreader: Llm-aided hardware test generation," in *2024 IEEE LLM Aided Design Workshop (LAD)*, 2024, pp. 1–5.

[8] L. Chen et al., "Software-based self-testing methodology for processor cores," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 3, pp. 369–380, 2001.

[9] M. Psarakis et al., "Microprocessor software-based self-testing," *IEEE Design & Test of Computers*, vol. 27, no. 3, pp. 4–19, 2010.

[10] D. Gizopoulos et al., "Systematic software-based self-test for pipelined processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 11, pp. 1441–1453, 2008.

[11] E. Sanchez et al., "On the functional test of branch prediction units," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 9, pp. 1675–1688, 2014.

[12] S. Yao et al., "React: Synergizing reasoning and acting in language models," in *The eleventh international conference on learning representations*, 2022.

[13] S. Di Carlo et al., "Software-based self-test of set-associative cache memories," *IEEE Transactions on Computers*, vol. 60, no. 7, pp. 1030–1044, 2010.

[14] K. Laeufer et al., "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[15] S. Canakci et al., "Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 529–534.

[16] O. Group, *Openhw group cv32e40p user manual*, Version 1.8.3, 2024.

[17] M. e. a. Gautschi, "Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[18] E. Kaja et al., "An automated and effective approach for sbst generation targeting risc-v cpus," in *2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, IEEE, 2024, pp. 1–4.

[19] R. Cantoro et al., "Assessing the effectiveness of software-based self-test programs for static cell-aware test," in *2024 IEEE European Test Symposium (ETS)*, IEEE, 2024, pp. 1–4.