REASSESSING SOFTWARE INTERMEDIATES IN SPEC-TO-RTL GENERATION: ACCURACY, COST, AND FAILURE BOUNDARIES

Anonymous authors
Paper under double-blind review

ABSTRACT

Writing correct register-transfer-level (RTL) designs in hardware description languages (HDLs) such as Verilog requires substantial engineering effort. High-level synthesis (HLS) offers a higher-level alternative by compiling behavioral descriptions in languages such as C and C++ into RTL. As modern models become substantially better at direct RTL generation, we ask whether these software relays still justify their added calls and translation boundaries. We study this question by separating the end-to-end *pipeline effect* of using a software relay from the *incremental artifact effect* of adding a software view while holding the final hardware rules and authoritative contract fixed. Across three repeats on 156 VerilogEval v2 and 50 RTLLM tasks, direct RTL achieves the highest observed pass rates (88.2% and 58.0%) and lowest token cost among eight evaluated pipelines. In a separate 100-task, three-repeat control, adding frozen Python/C++ views alongside an intact hardware contract yields no observed accuracy gain while increasing final-lowering token use by 18.8%/35.6%, excluding view construction. Controlled ablations and selected failure traces examine intermediate generation, execution, and RTL lowering, complemented by an analytical framework explaining why intermediate test success does not guarantee final RTL correctness. Together, these findings support direct RTL as the reference correctness–cost baseline in the evaluated module-level settings.

1 INTRODUCTION

Writing correct register-transfer-level (RTL) designs in hardware description languages (HDLs) such as Verilog requires substantial engineering effort. High-level synthesis (HLS) addresses this burden by compiling behavioral descriptions in languages such as C and C++ into RTL. Spec-to-RTL systems based on large language models (LLMs) adopt a related staged approach: they generate pseudocode, Python, or C++ as intermediate artifacts, followed by model-mediated translation or HLS-based lowering of suitable C/C++ code (Swaroopa et al., 2024; Yu et al., 2025). Beyond behavioral specification, intermediate artifacts can expose hierarchy and architectural decisions (DeLorenzo et al., 2025; Zuo et al., 2026; Islam et al., 2026). Recent work further develops these artifacts into typed, executable, or auditable design contracts (Yin et al., 2026b; Kong et al., 2026; Cheng et al., 2026; Yin et al., 2026a).

As LLMs become increasingly capable of generating RTL directly, the value of these software relays warrants reassessment. Ordinary software representations do not automatically preserve the target hardware's clock-edge behavior, non-blocking assignment semantics, cycle latency, reset sequencing, concurrency, and bit-width constraints (IEEE, 2018). Model-mediated lowering must preserve or recover these semantics, introducing additional opportunities for error. The utility of an intermediate layer is therefore *capability-relative*: when direct RTL generation is viable, its structure, verification, or control benefits must outweigh both the risk of semantic loss and the additional inference cost.

Prior work suggests that intermediate abstractions do not necessarily benefit capable LLMs. In AoT's o3-mini ablation on VerilogEval-Human, no evaluated abstraction configuration exceeds the direct baseline in reported functional pass@1, while retaining the base description alongside an IR

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

or pseudocode substantially narrows the deficit (DeLorenzo et al., 2025). Existing systems often introduce intermediate representations together with agents, retrieval, validation, or compiler support, making the contribution of the artifact difficult to isolate. We therefore examine whether complete software relays, including local validation and repair where available, improve the correctness–cost tradeoff, and whether their artifacts add value when the final generator retains the same hardware rules and authoritative contract.

We distinguish two complementary estimands. The *pipeline effect* compares direct RTL generation with seven frozen relay policies, including their stage-specific instructions, calls, and repairs. For each task, all policies share the same specification-derived `StructuredInfo` contract at entry, LLM configuration, and final evaluator; the main study covers 206 tasks with three repeats. The *incremental artifact effect* measures the contribution of appending a frozen Python or C++ view when the final hardware rules and authoritative contract are held fixed. A separate pre-specified control evaluates this effect on 100 primary tasks with three repeats. Homogeneous and matched-rule controls probe alternative explanations involving chain depth and final-rule wording. Robustness checks cover task difficulty, 131 tasks from two additional benchmarks, and three LLM configurations, while selected three-trace comparisons characterize observed failure patterns.

Direct RTL achieves both the highest functional pass rate and the lowest downstream token cost on the two main benchmarks, with the aggregate extension results broadly supporting this pattern. Matching final rules does not close the observed relay-only gap. In the separate persistent-contract control, observed pass rates differ from the contract-only baseline by less than one percentage point, while optional software views increase token use in the controlled final RTL generation calls by 19–36%, excluding view construction. The adjusted C++ interval excludes a gain of three percentage points or more, while Python remains unresolved at that margin. The two controls differ in their task samples and final prompt envelopes, so their results do not identify the effect of contract omission. Selected traces include cases in which the IR agrees with the reference on sampled inputs but the generated RTL disagrees or cannot run. Recorded initial-IR self-test passes likewise do not ensure final RTL correctness. Together, these findings support direct RTL as the reference baseline in the evaluated module-level setting, while requiring intermediate artifacts to demonstrate a hardware-relevant benefit that justifies their additional cost.

Our contributions are as follows:

- **Direct RTL generation outperforms the evaluated software relays.** Across 206 tasks and three repeats, direct LLM-based RTL generation achieves the highest observed pass rates (88.2% on VerilogEval v2 and 58.0% on RTLLM) and lowest downstream token cost among eight policies, providing a stronger correctness–cost baseline in the evaluated module-level setting.
- **Detailed analysis of intermediate-stage failures.** We combine controlled ablations, a persistent-contract experiment, and selected three-way traces to examine IR construction, local execution, and RTL lowering. These analyses distinguish pipeline effects from artifact value, assess translator defects, and localize observed failures across representation boundaries.
- **Intermediate success does not guarantee final RTL correctness.** Across both main benchmarks, 27.8–58.3% of initially self-test-passing IRs in P2 and P6 still lead to final RTL failures. Selected traces also identify IRs that match sampled reference behavior but yield incorrect RTL, demonstrating that local checks cannot replace end-to-end hardware evaluation.

2 Background and Related Work

Intermediate representations in spec-to-RTL generation. Intermediate artifacts play three roles in LLM-based RTL generation. As a capability bridge, they let a model write in a familiar software language and rely on HLS or a further translation step to reach RTL, as in annotated-C++ generation and pseudocode–Python–C++ refinement chains (Swaroopa et al., 2024; Yu et al., 2025). This rationale assumes that software generation is more reliable than direct RTL generation and that lowering preserves the intended behavior. As a reasoning scaffold, they make architectural decisions explicit before coding. SysVCoder and VeriGraphi expose hierarchy, interfaces, and connections (Zuo et al.,

2026; Islam et al., 2026), and AoT evaluates a circuit IR and pseudocode with and without the base description (DeLorenzo et al., 2025). As a design contract, they carry explicit hardware intent that can be checked. VeriRefine audits per-signal transition functions, HINT makes hardware intent executable with explicit lowering obligations, and CPPL validates a circuit IR before compiler-based lowering (Kong et al., 2026; Cheng et al., 2026; Yin et al., 2026b).

Evidence on intermediate value. The closest prior evidence comes from the o3-mini ablation of AoT on VerilogEval-Human. The direct base prompt reaches 74.6% functional pass@1, the IR alone 43.6%, base plus IR 74.0%, and base plus pseudocode 74.4% (DeLorenzo et al., 2025). Retaining the base description thus closes most of the deficit, but no abstraction exceeds direct generation. These abstractions are not executed, however, and the contract-based systems above add hardware constraints, checks, or compiler support that ordinary software intermediates lack. Neither line of work establishes whether an executable Python or C++ intermediate, checked locally and lowered by the model, improves final correctness enough to justify its cost.

Risks at representation boundaries. Each model-mediated transition is an opportunity for interface drift or semantic error. Ordinary Python and C++ do not enforce clocked state updates, cycle latency, reset behavior, concurrency, or bit-width and signedness rules, which hardware-oriented languages and frameworks encode explicitly (Bachrach et al., 2012; Izraelevitz et al., 2017; Jiang et al., 2020). A passing software test is therefore evidence about the tested behaviors only. It does not show that the artifact captures the required hardware semantics or that lowering preserves them. Differential testing can expose disagreement between executable implementations (McKeeman, 1998; Yang et al., 2011), but applying it across software and RTL requires a defined correspondence between their observable behaviors. Because published pipelines change representation, prompts, feedback, and lowering together, we separate the end-to-end pipeline effect from the incremental effect of a software view supplied beside a fixed, authoritative contract.

Research questions. These considerations lead to three questions. **RQ1 (pipeline value).** How do complete software-relay pipelines compare with direct RTL generation in final functional correctness and downstream inference cost? **RQ2 (artifact value and robustness).** Does the relay deficit persist under matched final rules, and does a software view add value when the authoritative contract is retained? How do these comparisons vary across task difficulty, benchmarks, and model configurations? **RQ3 (mechanism).** Where do relay failures arise across intermediate generation, execution, and lowering, and how often does a passing local self-test precede a failing final RTL?

3 METHODOLOGY

We separate the value of a complete relay from the incremental value of its artifact, using shared task inputs and final evaluation.

3.1 PROBLEM FORMULATION

Each task j consists of a natural-language specification s_j and an official evaluator E_j withheld from generation and repair. Two shared preprocessing calls produce a `StructuredInfo` contract $z_j = U(s_j)$ containing the interface, behavior, state, and hardware constraints, and frozen spec-derived tests $v_j = V(s_j)$. Both are computed once per task and reused across policies and repeats. Policy P_i contains L_i stages executed by the same model G_θ, including final RTL generation. Starting from $a^{(0)}_{j,i,r} = z_j$ in repeat r,

$$a^{(\ell)}_{j,i,r} = \mathcal{S}_\theta\big(a^{(\ell-1)}_{j,i,r};\, \tau_{i,\ell},\, v_j\big), \qquad \ell = 1, \ldots, L_i, \tag{1}$$

with final RTL $h_{j,i,r} = a^{(L_i)}_{j,i,r}$. The template $\tau_{i,\ell}$ specifies instructions, artifact fields, and permitted metadata. The operator $\mathcal{S}_\theta$ makes a generation call and, at executable stages, performs local checking and up to two repair calls, which do not count toward L_i. Runs that produce no RTL count as failures.

Two estimands. The *pipeline effect* compares final correctness and downstream cost between P0 and a complete relay under a shared entry contract, model, task set, and evaluator. It includes policy-specific prompts, payloads, and repairs, so it is not a prompt-invariant representation effect. The *incremental artifact effect* measures whether a frozen software view improves final generation when

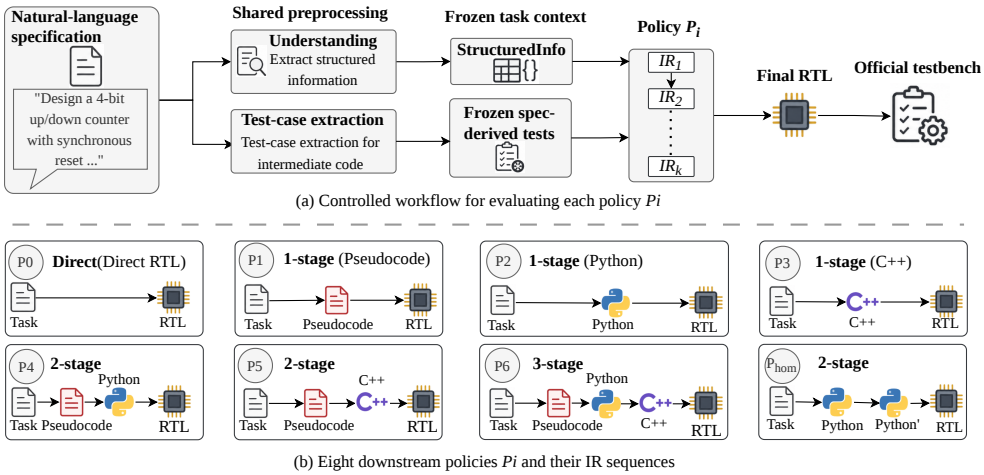


Figure 1: Shared preprocessing and eight downstream policies. `Task` denotes the frozen `StructuredInfo` contract. Stage labels count intermediates; L_i also includes final RTL generation. Executable stages receive local checking and bounded repair; official evaluation applies only to final RTL.

the hardware rules, authoritative contract, and source-precedence instructions are byte-identical. The persistent-contract control estimates this effect for Python and C++ views, excluding their construction cost (Section 4.2.2).

3.2 Information Boundary

Only shared preprocessing reads the raw specification. Direct generation and first-stage IR construction receive `StructuredInfo`; plan-to-RTL and plan-to-C++ receive typed plan fields, whereas plan-to-Python receives only logic steps. Python-to-RTL receives the module name and Python artifact, and C++-to-RTL receives the C++ artifact. These payload choices belong to the complete-policy intervention; the main comparison does not isolate contract omission from representation choice.

The implemented prompts also differ. Direct and C++-to-RTL prompts list hardware rules, while Python-to-RTL specifies the module name and output format. The C++ template contains five rules referring to an unavailable specification. Matched-rule, orphan-reference, and persistent-contract controls examine distinct restricted comparisons (Section 4.2.2; Appendix C). No downstream call receives the raw specification, the full frozen test suite, or the official testbench. The official evaluator scores each final RTL once and never guides repair or candidate selection. Exact prompts appear in Appendix D.

3.3 Representation Policies

P0 maps `StructuredInfo` directly to RTL. P1 inserts a non-executable pseudocode plan; P2 and P3 use Python and C++, respectively; P4 and P5 prepend a plan to those executable intermediates; and P6 traverses pseudocode, Python, and C++ (Figure 1). The homogeneous control P_{hom} follows `StructuredInfo` $\rightarrow$ Python $\rightarrow$ Python$'$ $\rightarrow$ RTL. Its second Python call is instructed to preserve behavior, interfaces, widths, and clock/reset semantics. Comparing it with P2 probes an additional same-language transition, while heterogeneous policies also change representation family.

3.4 Executable-IR Validation and Blind Repair

Each executable Python or C++ stage runs against v_j. Following a failed check, up to two blind repair calls receive a bounded IR prefix and failure report, including expected and observed values

when available. They receive neither the full tests nor the official testbench. In P6, Python is checked before C++ translation, and C++ before RTL lowering. In P_{hom}, the initial Python artifact is checked before reserialization. Pseudocode receives no execution-based check.

Python uses a class-based interface with explicit state updates. The C++ harness supports step methods, fixed-width values, nested outputs, default controls, and implicit clocks. Frozen input–output tests are extracted from the first 4,000 specification characters. These checks provide evidence about selected software behaviors; their relation to final correctness is formalized next.

3.5 Analytical View of Relay Utility

We analyze relay correctness through a probabilistic decomposition, clarify the limits of local self-test acceptance, and derive the accuracy threshold for matching a baseline on tokens per pass. Fix a policy, a task distribution, and an executable IR checkpoint. Let $I = 1$ denote IR correctness under a specified interpretation as hardware input/output traces, and $R = 1$ final RTL correctness under the same target contract, including applicable timing, reset, state, and width requirements. Missing or non-executable artifacts have correctness zero. Probabilities average over tasks and generation randomness.

Define $q = \Pr(I = 1)$, $d = \Pr(R = 0 \mid I = 1)$, and $\rho = \Pr(R = 1 \mid I = 0)$. For relay correctness $p_R = \Pr(R = 1)$ and direct correctness p_0 under the same distribution and criterion,

$$\begin{aligned} p_R &= q(1 - d) + (1 - q)\rho, \\ p_R - p_0 &= (q - p_0) - qd + (1 - q)\rho. \end{aligned} \tag{2}$$

The terms account for checkpoint correctness relative to direct generation, failure following a correct IR, and success following an incorrect IR. They allow recovery but neither identify causal contributions nor imply a monotonic depth penalty.

Let $\mathcal{A}$ denote recorded local self-test acceptance at this checkpoint, with $\Pr(\mathcal{A}) > 0$. Define $q_{\mathcal{A}} = \Pr(I = 1 \mid \mathcal{A})$, $d_{\mathcal{A}} = \Pr(R = 0 \mid I = 1, \mathcal{A})$, and $\rho_{\mathcal{A}} = \Pr(R = 1 \mid I = 0, \mathcal{A})$. Then

$$\Pr(R = 0 \mid \mathcal{A}) = q_{\mathcal{A}} d_{\mathcal{A}} + (1 - q_{\mathcal{A}})(1 - \rho_{\mathcal{A}}). \tag{3}$$

Thus, final failure after local acceptance can follow either a correct or an incorrect IR. For P6's initial Python checkpoint, downstream terms cover the remaining translations, checks, and repairs. Appendix B.3 gives a sufficient guarantee under bounds on residual IR error and downstream failure.

The empirical endpoint is official-test success Y, using the outcome notation below with indices suppressed. Replacing R by Y and redefining the downstream conditional terms gives the corresponding finite-test decomposition. The reported rates and selected traces do not identify the semantic parameters above.

Token efficiency. For mean counted tokens $c_i > 0$ and observed benchmark pass fractions $\pi_i > 0$ on the same task set, matching a baseline on tokens per pass requires

$$\frac{c_1}{\pi_1} \le \frac{c_0}{\pi_0} \quad \Longleftrightarrow \quad \pi_1 \ge \pi_0 \frac{c_1}{c_0}. \tag{4}$$

This is a threshold for the stated scalar metric at fixed mean costs; it does not determine Pareto dominance.

3.6 Failure-Boundary Localization

We compare stored software, reference-RTL, and generated-RTL traces on shared sampled inputs. The executor supports combinational behavior, so the audit covers failed VerilogEval P2/P3 runs from repeat `r0`, excluding tasks whose `StructuredInfo` declares state or clock/reset configuration. It distinguishes IR execution failure, IR–reference disagreement, and RTL disagreement or execution failure after sampled IR–reference agreement. Cases without a traceable reference remain unclassified. These labels describe observed disagreements; sampled agreement does not establish equivalence or identify the call that introduced an error.

Separately, for P2 and P6, we report official-test failure conditional on recorded initial-IR self-test acceptance. This statistic instantiates the finite-test version of Eq. 3; it does not isolate a lowering-error rate. Results appear in Sections 4.2.2 and 4.2.3, with details in Appendix B.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Benchmarks. The main study uses 156 VerilogEval v2 tasks covering combinational and sequential logic, arithmetic, finite-state machines, and protocols (Pinckney et al., 2024), and 50 RTLLM tasks with official self-checking testbenches (Lu et al., 2024). Three repeats give 468 VerilogEval and 150 RTLLM runs per policy. A three-repeat supplement adds 10 complete modules from the AES, RISC-V, and SD-card IPs of RealBench (Jin et al., 2025) as a small stress test on modules drawn from larger designs. To check whether the comparison generalizes, we run one repeat on 71 ArchXBench tasks (Purini et al., 2025), grouped by their ordered native levels into 26 Easy, 24 Medium, and 21 Hard tasks, and on 60 public CVDP code-generation tasks (Pinckney et al., 2025) with 32 Easy and 28 Medium labels.

Model and protocol. All main-study calls use GPT-5.4 at low reasoning effort with an 8,192-token output cap, no fallback model, and seeds 1103, 2207, and 3301 for the three repeats. The P0 prompt was reduced from 796 to 132 lines by removing six worked examples and redundant rules while keeping the interface and hardware constraints. Final RTL is syntax checked and run once against the official testbench. VerilogEval, RTLLM, ArchXBench, and CVDP use their supplied harnesses, and RealBench uses Verilator. Generation failures, malformed or non-executable RTL, timeouts, and functional mismatches count as failures, while infrastructure failures are resumed rather than scored. Prompt and artifact hashes, usage events, retries, and verifier results are logged for every run.

Robustness and fairness controls. A prespecified static complexity score stratifies the 216 three-repeat tasks into 133 Easy, 74 Medium, and 9 Hard tasks, and each relay is compared with P0 within every benchmark and stratum and after pooling benchmarks within a stratum. A stratified 40-task subset repeats P0, P1, P2, and P6 once under GPT-5.4, GPT-5.4-mini, and GPT-4o-mini. A strict routing gate rejects any response served by a different model, and we treat the three configurations as labels rather than an ordered capability scale.

Two further controls examine the relay deficit directly. The matched-rule control gives three final RTL calls a byte-identical hardware-rule prefix and varies only the source. The source is `StructuredInfo` for A0, a frozen Python artifact for A2, and a frozen C++ artifact for A3. Its primary scope is a 20-task VerilogEval subset within a 40-task pool. Because A2 and A3 still lack the contract, this control tests rule asymmetry but not contract access.

The persistent-contract control tests the incremental artifact effect. Its protocol, manifest, benefit margin, and prompts were hash-frozen before any API call. Its primary scope is 100 tasks unused in an earlier pilot (75 VerilogEval and 25 RTLLM), with 15 ArchXBench Hard tasks reported separately. PC0 receives the contract alone, while PC2 and PC3 add a frozen Python or C++ view. All three conditions share byte-identical rules and a source-precedence instruction that declares the contract authoritative. Each condition runs three repeats under strict routing, with rotated condition order and no RTL repair.

Metrics. The outcome $Y_{j,i,r}$ is one when the final RTL passes the official test. In the main study, $C_{j,i,r}$ sums recorded input and output tokens over policy-specific generation, translation, and repair calls, excluding shared preprocessing. In the persistent-contract control, cost covers only final RTL calls and formatting retries, excluding view construction.

We report pass rate, mean tokens per run, and tokens per passing solution; the last divides total counted tokens by the number of passing runs and includes spending on failures. Budgeted success $Q(P_i; B)$ is the fraction of runs that pass while costing at most B, for $B \in \{16\text{k}, 32\text{k}, 48\text{k}\}$, and the Software-IR Tax is $T_Q(P_i; B) = Q(P_0; B) - Q(P_i; B)$. These summaries threshold completed runs; they do not implement budget-adaptive generation or candidate selection. The normalized tax area appears in Appendix C.4.

Statistical analysis. Comparisons are paired by task. We average repeats within each task and policy and form 95% confidence intervals for paired differences from 10,000 task-level bootstrap samples, which preserves the correlation among repeats of the same task. The one-repeat controls use exact paired McNemar tests with Holm correction within prespecified families. For the two persistent-contract contrasts, we compare the upper endpoints of Bonferroni-adjusted 97.5% bootstrap intervals

Table 1: Three-repeat main results. N counts task–repeat runs per policy. Tokens per pass divides total downstream tokens by passing runs, including spending on failures.

	VerilogEval v2 ($N = 468$)		RTLLM ($N = 150$)	
Policy	Pass	Tokens/pass	Pass	Tokens/pass
P0	**88.2%**	**7,960**	**58.0%**	**12,666**
P1	69.9%	16,991	52.7%	24,405
P2	50.4%	31,542	48.0%	39,899
P3	33.5%	39,410	33.3%	41,013
P4	39.5%	59,349	38.0%	69,573
P5	52.1%	31,644	50.7%	34,100
P6	32.9%	94,424	35.3%	95,661
P_{hom}	46.6%	45,617	42.0%	58,955

with a prespecified benefit margin of three percentage points. This tests whether such a benefit can be excluded but does not establish equivalence. A tie in a directional summary means equal point estimates. Results on RealBench and the Hard strata are treated as directional.

4.2 Results

The results address the three contributions in order: the correctness–cost advantage of direct RTL, controlled analysis of intermediate-stage failures, and the limits of intermediate self-test success.

4.2.1 Main Correctness–Cost Frontier

P0 achieves the highest observed correctness and lowest downstream cost on both main benchmarks (Table 1, Figure 2). It passes 88.2% of VerilogEval runs and 58.0% of RTLLM runs at about 7k tokens per run. P1 trails by 18.4 and 5.3 percentage points while spending about twice as many tokens per pass. P6 consumes 4.4–4.6 times P0's tokens per run, with pass rates of 32.9% and 35.3%. Every relay has positive Software-IR Tax at each evaluated budget.

Depth alone does not determine the ranking: P5 outperforms P2 on both benchmarks, while P_{hom} stays closer to P2 than P6 does. This is consistent with Eq. 2, which permits both loss and recovery without imposing a depth ordering. The experiment estimates complete-policy effects, not the individual terms of that decomposition.

The frontier remains after excluding every policy using the audited C++ translator. P1, P2, P4, and P_{hom} all remain below P0 in observed accuracy and above it in cost. At $B = 48\mathrm{k}$, all four paired intervals for $Q(P_i; B) - Q(P_0; B)$ are negative on VerilogEval; on RTLLM, two are negative and the other two have negative point estimates.

The broader checks support the direction with limited precision. All 21 pooled difficulty contrasts are negative; among 63 benchmark-specific contrasts, 51 are negative, 11 tie, and one is positive with an interval including zero. On 131 extension tasks, 33 of 35 contrasts are negative; the two positive contrasts involve one or two tasks and have Holm-adjusted $p = 1.0$. All nine pooled model-configuration contrasts are negative, but interaction intervals include zero, so these runs do not establish a capability-scaling relation. On the 10-task RealBench supplement, P0 and P1 tie at 43.3%, with P0 using about 40% fewer tokens. Detailed extensions appear in Appendix C.

4.2.2 Controlled Analyses and Intermediate Failure Localization

Persistent-contract comparison. In the separate 100-task control, PC0, PC2, and PC3 pass 79.0%, 78.3%, and 78.7%, respectively. Paired differences from PC0 are -0.67 points (95% CI $[-4.33, +3.00]$) for Python and -0.33 points ($[-3.00, +2.00]$) for C++. The adjusted upper endpoints exclude a three-point benefit for C++ ($+2.33$) but not Python ($+3.67$); neither contrast establishes harm or equivalence. Optional views increase final-call tokens by 18.8% and 35.6%, excluding view construction. Neither improves the observed correctness–cost frontier (Figure 3; Appendix C.7).

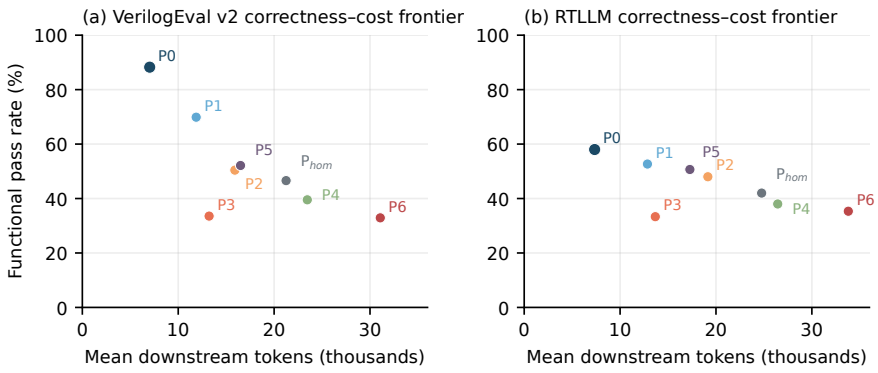


Figure 2: Three-repeat correctness–cost comparisons. Points show functional pass rate against mean downstream tokens per run. P0 has the highest observed pass rate and lowest cost on each benchmark.

Task-level win/loss/tie counts are 4/6/90 for Python and 2/2/96 for C++, so the near-zero aggregate contrasts coexist with changes on individual tasks. The control estimates the value of the specified frozen views under common final instructions; it does not evaluate a complete relay carrying the contract through every stage.

Equation 4 translates these costs into a descriptive threshold. With $\pi_0 = 0.790$, $c_0 = 1425.6$, $c_2 = 1694.3$, and $c_3 = 1933.2$, Python requires approximately 93.9% pass rate to match PC0's tokens per pass, a 14.9-point gain. C++ would require 107.1%, making parity on this scalar metric unattainable at its observed mean final-call cost. These thresholds fix the measured costs and make no claim about Pareto efficiency at other accuracy–cost combinations.

Translator and rule controls. The additional controls address narrower alternatives. Matching final rules leaves a 30-point relay-only deficit on 20 VerilogEval tasks (adjusted $p = 0.094$); the secondary 40-task pool shows deficits of 22.5 points for Python and 20.0 points for C++ (both adjusted $p = 0.023$). On 60 tasks with three repeats, rewriting five orphaned C++ rules leaves both original and revised prompts at 37.2% (adjusted 97.5% difference CI $[-4.4, +4.4]$), versus 82.8% for a contemporaneous direct arm. This local edit does not restore parity and does not resolve other prompt defects.

These controls establish separate observations: relay-only deficits under matched rules, and near-zero auxiliary-view contrasts beside a fixed contract. Their different task samples and prompt envelopes prevent attributing the main-study deficit to contract omission.

Observed failure boundaries. The three-trace audit examines 88 selected failed combinational VerilogEval runs: 28 P2 and 60 P3. In P2, 57% show IR–reference disagreement or IR execution failure. In P3, 52% have sampled IR–reference agreement followed by RTL disagreement or execution failure (Table 2). These observations locate disagreements in stored artifacts; they neither establish complete IR correctness nor estimate population transition probabilities.

Existing task stratification adds a descriptive contrast. On VerilogEval, P0 passes 88.9% of combinational and 87.6% of sequential tasks, while P1 passes 86.0% and 52.4%. P3 has the opposite ordering, at 25.1% and 42.7%. These differences show that relay behavior varies by policy and task type; they do not isolate reset semantics or another specific cause.

A separate RTLLM comparison distinguishes syntax from functional success: P0 and P6 both achieve 98.0% syntax pass, while functional pass rates are 58.0% and 35.3%. Together, these observations support evaluating final hardware behavior alongside local software checks. Detailed failure summaries and task-stratum results appear in Appendix B.

4.2.3 Local Self-Test Success and Final RTL Correctness

Among recorded initial-IR self-test passes, final RTL still fails in 27.8%/58.3% of VerilogEval P2/P6 runs and 31.1%/50.0% on RTLLM (Figure 4). For P6, acceptance refers to the initial Python check-

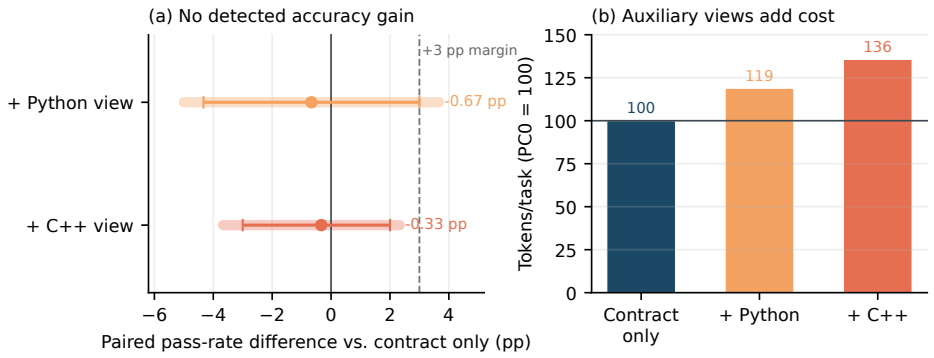


Figure 3: Persistent-contract effects. (a) Paired pass-rate differences with 95% and Bonferroni-adjusted 97.5% intervals; the dotted line marks the three-point benefit margin. (b) Mean final-call tokens per run, indexed to PC0 and excluding view construction.

point. As the finite-test version of Eq. 3 makes explicit, this statistic combines final failure following a correct IR with final failure following an incorrect but locally accepted IR. Each policy conditions on its own accepted runs; the higher P6 rate does not identify a causal depth effect.

The trace cases in Section 4.2.2 likewise include IRs that agree with the reference on sampled inputs but yield disagreeing or non-executable RTL. Such agreement remains finite-test evidence. Appendix B.3 gives one sufficient route to an end-to-end guarantee by bounding residual error among accepted IRs and subsequent failure following a correct IR; the present measurements do not certify those bounds.

Connection to the contributions. Section 4.2.1 supports the first contribution by establishing direct RTL's observed correctness–cost advantage among the evaluated pipelines and examining its robustness. Section 4.2.2 supports the second contribution by distinguishing pipeline effects from incremental artifact value, assessing specific translator defects, and locating disagreements among selected intermediate and RTL artifacts. Section 4.2.3 supports the third contribution by showing that recorded local self-test success does not certify final RTL correctness and interpreting this distinction through the analytical framework.

5 Conclusion

Across the main module-level benchmarks and implemented policies, direct RTL generation from the shared hardware contract achieves the highest observed functional correctness at the lowest downstream token cost. In the persistent-contract control, adding frozen software views does not improve observed accuracy and increases final-call cost. The failure analysis further shows that passing a local software self-test is insufficient to ensure final RTL correctness. These findings support direct RTL as a reference baseline in the evaluated setting and motivate assessing each intermediate representation through the contribution of its structure, checking, and lowering mechanism to the final correctness–cost tradeoff.

AI Use Statement

Generative AI tools were used only for language polishing during manuscript preparation. Their use as systems under study is described in the methodology. The authors reviewed the final text and take responsibility for the manuscript.

References

Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded lan-

guage. In *Proceedings of the 49th Annual Design Automation Conference (DAC)*, pp. 1216–1225, 2012. doi: 10.1145/2228360.2228584.

Tairan Cheng, Yi Liu, Dongsheng Zuo, Zhengyuan Shi, Hongji Zhang, Xiangfei Hu, Maoshuo He, Hao Yan, and Qiang Xu. HINT: Toward an executable hardware-intent representation layer for llm-driven rtl generation. *arXiv preprint arXiv:2608.07625*, 2026. doi: 10.48550/arXiv.2608.07625.

Matthew DeLorenzo, Kevin Tieu, Prithwish Jana, Piyush Jha, Dileep Kalathil, Vijay Ganesh, and Jeyavijayan Rajendran. Abstractions-of-thought: Intermediate representations for llm reasoning in hardware design. *arXiv preprint arXiv:2505.15873*, 2025. doi: 10.48550/arXiv.2505.15873.

IEEE. IEEE Std 1800-2017: IEEE standard for SystemVerilog—unified hardware design, specification, and verification language, 2018. URL https://standards.ieee.org/ieee/1800/6700/.

Sazzadul Islam, Tasnim Tabassum, and Hao Zheng. Verigraphi: A multi-agent framework of hierarchical rtl generation for large hardware designs. *arXiv preprint arXiv:2604.14550*, 2026. doi: 10.48550/arXiv.2604.14550.

Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 209–216, 2017. doi: 10.1109/ICCAD.2017.8203780.

Shunning Jiang, Peitian Pan, Yanghui Ou, and Christopher Batten. PyMTL3: A python framework for open-source hardware modeling, generation, simulation, and verification. *IEEE Micro*, 40(4): 58–66, 2020. doi: 10.1109/MM.2020.2997638.

Pengwei Jin, Di Huang, Chongxiao Li, Shuyao Cheng, Yang Zhao, Xinyao Zheng, Jiaguo Zhu, Shuyi Xing, Bohan Dou, Rui Zhang, Zidong Du, Qi Guo, and Xing Hu. Realbench: Benchmarking verilog generation models with real-world ip designs, 2025. URL https://arxiv.org/abs/2507.16200.

Xiangfei Kong, Tasnim Tabassum, Marwan Abdelwahab, and Hao Zheng. A progressive approach to synthesizable rtl design generation using llms. *arXiv preprint arXiv:2607.18519*, 2026. doi: 10.48550/arXiv.2607.18519.

Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. Rtllm: An open-source benchmark for design rtl generation with large language model. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024. URL https://arxiv.org/abs/2308.05345.

William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. Revisiting VerilogEval: Newer LLMs, in-context learning, and specification-to-RTL tasks, 2024. URL https://arxiv.org/abs/2408.11053v1.

Nathaniel Pinckney, Chenhui Deng, Chia-Tung Ho, Yun-Da Tsai, Mingjie Liu, Wenfei Zhou, Brucek Khailany, and Haoxing Ren. Comprehensive verilog design problems: A next-generation benchmark dataset for evaluating large language models and agents on RTL design and verification. *arXiv preprint arXiv:2506.14074*, 2025. doi: 10.48550/arXiv.2506.14074. URL https://arxiv.org/abs/2506.14074.

Suresh Purini, Siddhant Garg, Mudit Gaur, Sankalp Bhat, Sohan Mupparapu, and Arun Ravindran. ArchXBench: A complex digital systems benchmark suite for LLM-driven RTL synthesis. In *Proceedings of the ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD)*, 2025. URL https://arxiv.org/abs/2508.06047.

Sneha Swaroopa, Rijoy Mukherjee, Anushka Debnath, and Rajat Subhra Chakraborty. Evaluating large language models for automatic register transfer logic generation via high-level synthesis. *arXiv preprint arXiv:2408.02793*, 2024. doi: 10.48550/arXiv.2408.02793.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 283–294, 2011. doi: 10.1145/1993498.1993532.

Chenyang Yin, Agasthi Haputhanthri, Aditya Anirudh Jonnalagadda, Zhenyu Bai, Yuanming Song, Saranyu Chattopadhyay, Mohammad Fadiheh, Tom Zelazny, Subhasish Mitra, and Tulika Mitra. Llm-based hardware development with hierarchical irs and end-to-end multi-agent workflow. *arXiv preprint arXiv:2608.30659*, 2026a. doi: 10.48550/arXiv.2608.30659.

Shuo Yin, Yihe Wang, Lancheng Zou, Xufeng Yao, Tinghuan Chen, Chen Bai, Zhengrong Wang, Tsung-Yi Ho, and Bei Yu. CPPL: A circuit prompt programming language. *arXiv preprint arXiv:2605.17892*, 2026b. doi: 10.48550/arXiv.2605.17892.

Zhongzhi Yu, Mingjie Liu, Michael Zimmer, Yingyan Celine Lin, Yong Liu, and Haoxing Ren. Spec2RTL-Agent: Automated hardware code generation from complex specifications using LLM agent systems. *arXiv preprint arXiv:2506.13905*, 2025. doi: 10.48550/arXiv.2506.13905. URL https://arxiv.org/abs/2506.13905.

Jian Zuo, Junzhe Liu, Xianyong Wang, Chen Liang, Navya Goli, Umamaheswara Rao Tida, Zhenge Jia, Zhaoyan Shen, and Mengying Zhao. SysVCoder: An LLM-driven framework for systematic generation of system-level design. *arXiv preprint arXiv:2504.20653v3*, 2026. doi: 10.48550/arXiv.2504.20653. URL https://arxiv.org/abs/2504.20653v3. Version 3, revised 30 June 2026; first submitted 29 April 2025.

A Extended Discussion and Limitations

A.1 Interpreting the Two Estimands

The pipeline effect asks whether a complete implemented relay improves the end-to-end correctness–cost tradeoff. It includes the prompts, information boundaries, calls, and repairs of that policy. In the main study, every relay has lower observed correctness and higher downstream cost than P0 (Table 1).

The incremental artifact effect asks whether a frozen software view improves final generation given the same authoritative contract and hardware rules. On 100 primary tasks, the Python and C++ contrasts are within one percentage point of zero, while final-call cost increases (Appendix C.7). The adjusted interval excludes a three-point benefit for C++ and leaves Python unresolved at that margin. These observations establish neither harm nor equivalence.

The matched-rule and persistent-contract controls use different task samples and final prompt envelopes. They establish a relay-only deficit under matched rules and no observed accuracy gain from the tested views beside a fixed contract. They do not identify how much of the main-study gap is caused by contract omission, or whether a different artifact or lowering mechanism would help. The decomposition in Section 3.5 organizes possible outcomes without estimating their separate causal contributions.

A.2 Relation to Prior Evidence

Our results parallel the o3-mini ablation of AoT. There, an IR alone yields a substantial deficit, while supplying the base description alongside an IR or pseudocode largely closes it without exceeding the direct baseline (DeLorenzo et al., 2025).

Our study extends this comparison in two directions. First, the intermediates are executable and pass through local tests and bounded repair, which gives them an opportunity AoT's abstractions did not have. Second, the persistent-contract control isolates the value of a software view from the value of the pipeline that produced it.

The failure analysis illustrates why executability alone does not certify final correctness (Appendix B). Local tests inspect selected behaviors, while the final RTL must satisfy the hardware contract after subsequent transformations. The analytical framework allows both loss and recovery; Appendix B.3 states one sufficient condition for an end-to-end guarantee.

A.3 Design Implications

This analysis directs attention to the semantics and lowering mechanism of an intermediate representation, not to its language family.

Hardware-oriented languages and IRs assign explicit meaning to state, timing, concurrency, and bit widths (Bachrach et al., 2012; Izraelevitz et al., 2017; Jiang et al., 2020). Ordinary Python and C++ need additional modeling conventions and checks to express the same commitments and to carry them into RTL. Recent systems move in this direction. VeriGraphi exposes design structure through a specification graph, CPPL combines IR validation with compiler-based lowering, and HINT makes hardware intent executable with explicit obligations for RTL lowering (Islam et al., 2026; Yin et al., 2026b; Cheng et al., 2026).

We see three practical consequences for evaluating such proposals.

- An intermediate should state which hardware properties it makes explicit, how they are checked, and how they survive lowering.
- It should be compared against direct generation from the same contract, on both correctness and cost.
- Local checks should be reported together with end-to-end outcomes, because a local pass alone overstates final correctness.

Within the evaluated setting, direct generation from the shared contract is the natural reference baseline. The comparison should be repeated when the model, the task scope, or the available verification changes.

A.4 Limitations

Scope of the policies. Our claims concern the implemented module-level pipelines, including their prompts, information boundaries, and model-mediated lowering. They are not claims about software representations in general. The relays lower from the intermediate without the contract, which matches the IR-only condition of AoT. A relay that carries the contract through every stage was not evaluated as a complete pipeline.

Prompts and translators. The Python-to-RTL prompt specifies no hardware rules beyond the module name, whereas the direct prompt does. The C++-to-RTL prompt contains heuristics for signed casts and output timing that are not generally semantics-preserving. The matched-rule control and the orphan-reference ablation address two specific asymmetries, but they do not isolate the remaining heuristics or the quality of the upstream artifacts. A 6,000-character guard clipped 2 of 1,850 Python-to-RTL payloads (Appendix D).

Local tests.

The frozen tests are extracted by a model from the first 4,000 characters of the specification. Their coverage depends on the extracted cases and the software execution interface. We treat local acceptance as a measured signal rather than a certificate of complete hardware correctness. The reported conditional failure rate does not identify the residual IR error or downstream failure components separately.

Model and statistical power. The main study uses a single model at low reasoning effort. The configuration check covers three models from one provider with one repeat. Several extensions and controls use one repeat, RealBench contains only 10 tasks, and the Hard strata are small.

Trace analysis. The trace analysis covers selected failed combinational cases from one repeat. It does not estimate how often each failure boundary occurs across all tasks.

Outcomes and scope. Our outcomes are functional correctness and inference cost. We do not evaluate compiler-based HLS lowering, hardware-semantic IRs, formally verified artifacts, a direct baseline with RTL-level simulation and repair against the same frozen tests, or system-scale designs.

B Failure-Boundary Analysis

This section applies the two analyses defined in Section 3.6 to locate where relay failures arise.

B.1 Observed Failure Patterns

Each task is traced on a shared set of input vectors obtained by deduplicating 16 boundary-pattern and pseudorandom candidates. Percentages use the full selected set within each policy. Cases without a traceable reference remain unclassified and are omitted from the rows of Table 2.

The two policies fail at different boundaries.

In P2, 57% of the selected failures are visible in the software artifact. The Python trace disagrees with the reference in 43% of cases, and the Python model cannot run under the trace executor in 14%. A further 36% have a Python trace that matches the reference while the generated RTL cannot run under the executor. Of these ten cases, nine fail elaboration because an expected port name is absent from the generated RTL. This pattern is consistent with interface drift at the Python-to-RTL boundary, where the lowering prompt receives no port list.

In P3, 52% of the selected failures arise after an IR whose trace matches the reference. The RTL trace disagrees in 37% of cases and the RTL cannot run in 15%. These observations concern the selected P3 failures and do not identify which translator rules caused them.

The labels describe where the recorded artifacts diverge on sampled inputs. They do not identify which generation, repair, or lowering call introduced the error, and agreement on sampled inputs does not establish functional equivalence. Because the cases are selected failures from a single repeat, the table characterizes failure boundaries under the evaluated prompts rather than their prevalence.

Table 2: Observed failure patterns in 88 selected failed combinational VerilogEval v2 runs from repeat `r0` (28 from P2 and 60 from P3). Trace agreement and disagreement refer only to sampled inputs. Percentages use the full selected set within each policy, including cases without a traceable reference, which remain unclassified and are omitted from the rows.

Observed failure pattern	P2 (Python)	P3 (C++)
IR trace disagrees with the reference	43%	35%
IR cannot run under the trace executor	14%	8%
IR trace matches the reference, RTL trace disagrees	4%	37%
IR trace matches the reference, RTL cannot run under the trace executor	36%	15%

B.2 Local Self-Tests and Final Correctness

The second analysis relates recorded initial-IR self-test acceptance to final official-test outcomes. Conditional failure rates are 27.8% and 58.3% for P2 and P6 on VerilogEval v2, with 223 and 223 accepted runs, and 31.1% and 50.0% on RTLLM, with 45 and 50 accepted runs (Figure 4).

The finite-test version of Eq. 3 separates final failure following a correct IR from final failure following an incorrect but locally accepted IR. The aggregate statistic does not identify the two components. For P6, acceptance refers to the initial Python checkpoint, so the downstream terms include the remaining Python-to-C++ and C++-to-RTL process, with intervening checks and repairs. Policies condition on their own accepted runs; their difference does not isolate a causal effect of depth.

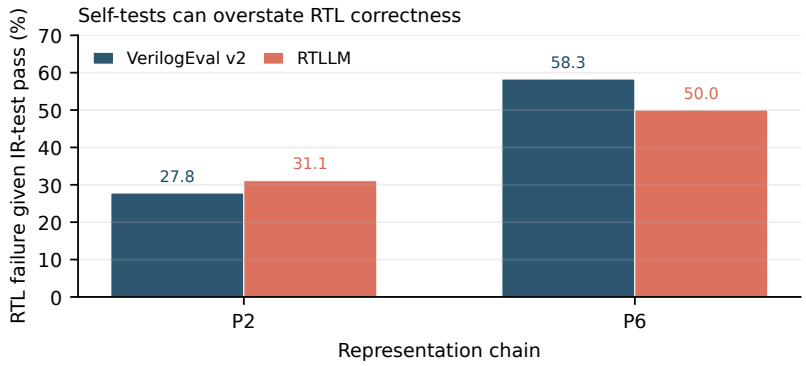


Figure 4: Final RTL failure conditional on recorded initial-IR self-test acceptance. P2/P6 denominators are 223/223 runs on VerilogEval v2 and 45/50 on RTLLM. For P6, the acceptance event refers to the initial Python checkpoint.

B.3 A Sufficient Bound after Local Acceptance

Use the semantic events in Section 3.5. Conditional terms on zero-probability branches do not affect the decompositions.

Proposition. Suppose $\Pr(\mathcal{A}) > 0$ and $\epsilon, \eta \in [0, 1]$ satisfy

$$\Pr(I = 0 \mid \mathcal{A}) \leq \epsilon, \qquad \Pr(R = 0 \mid I = 1, \mathcal{A}) \leq \eta,$$

where the second condition is required only if $\Pr(I = 1 \mid \mathcal{A}) > 0$. Then

$$\Pr(R = 0 \mid \mathcal{A}) \leq \epsilon + (1 - \epsilon)\eta. \tag{5}$$

Proof. Let $a = \Pr(I = 0 \mid \mathcal{A}) \le \epsilon$. Partitioning on I gives

$$\begin{aligned}\Pr(R = 0 \mid \mathcal{A}) &= \Pr(R = 0, I = 0 \mid \mathcal{A}) + \Pr(R = 0, I = 1 \mid \mathcal{A}) \\ &\le a + (1 - a)\eta \le \epsilon + (1 - \epsilon)\eta,\end{aligned}$$

since $\eta \le 1$. No independence assumption is required. □

Here ϵ bounds semantic error among accepted IRs, which differs from the acceptance probability $\Pr(\mathcal{A} \mid I = 0)$ on incorrect IRs. The proposition supplies a sufficient condition; the reported tests and selected traces do not certify numerical values for ϵ or η.

B.4 Combinational and Sequential Tasks

Stratifying tasks by combinational versus sequential or stateful features reveals where each relay loses accuracy. No relay has a statistically supported advantage over P0 in any main stratum.

On VerilogEval, P0 passes 88.9% of combinational and 87.6% of sequential tasks. P1 nearly matches P0 on combinational tasks at 86.0% but falls to 52.4% on sequential tasks.

The plan path is a plausible source of this collapse. The plan schema carries the clock and reset configuration, and the pseudocode prompt defaults to an active-low reset named `rst_n` when the contract does not supply one. The plan-to-RTL prompt then renders a synchronous reset style. Tasks with asynchronous or active-high resets are therefore exposed to a systematic mismatch on this path. We have not isolated this factor experimentally.

P3 shows the opposite pattern, passing 25.1% of combinational and 42.7% of sequential tasks. This agrees with the trace analysis, which places most combinational C++ failures at or after lowering. P5 passes 59.3% and 44.4% of the two groups.

On RTLLM, P0 passes 76.7% of combinational and 53.3% of sequential tasks, and P1 passes 76.7% and 46.7%. The RealBench split contains only two combinational tasks, so apparent gains there carry no inferential weight. None of its eight sequential tasks shows a relay advantage.

C Robustness Results and Control Experiments

Throughout this section, IR utility denotes a relay's pass rate minus the pass rate of P0 on the same tasks. Negative values therefore indicate a relay deficit.

C.1 Difficulty Strata and Model Configurations

Figure 5(a) reports pooled IR utility across the three difficulty strata of the three-repeat study. All 21 pooled contrasts are negative. Among the 63 benchmark-specific contrasts, 51 are negative, 11 are ties, and one is positive. The positive contrast is P1 on the seven-task RealBench-Medium subset, and its interval includes zero.

Deficits are smaller in the Hard stratum. This stratum contains only nine tasks, and P0 is near the floor there, which bounds how far any relay can fall below it. The smaller Hard deficits therefore do not establish a trend toward relay utility on harder tasks.

Figure 5(b) reports pooled IR utility on the 40-task sensitivity subset. Values below are percentage-point differences from P0; all nine pooled contrasts are negative.

Configuration	P1 (pp)	P2 (pp)	P6 (pp)
GPT-5.4	−10.0	−15.0	−30.0
GPT-5.4-mini	−7.5	−12.5	−20.0
GPT-4o-mini	−17.5	−12.5	−32.5

Of the 27 stratified configuration–policy contrasts, 22 are negative and five are ties. All cross-configuration interaction intervals include zero.

The strict router recorded and rejected substituted gateway responses before they could produce a verdict, so no outcome mixes models. The remaining limitations are the single repeat and the fact

that each configuration bundles a model with its provider settings. Because GPT-5.4-mini scores no lower than GPT-5.4 on the direct baseline of this subset, the configurations do not form an ordered capability scale.

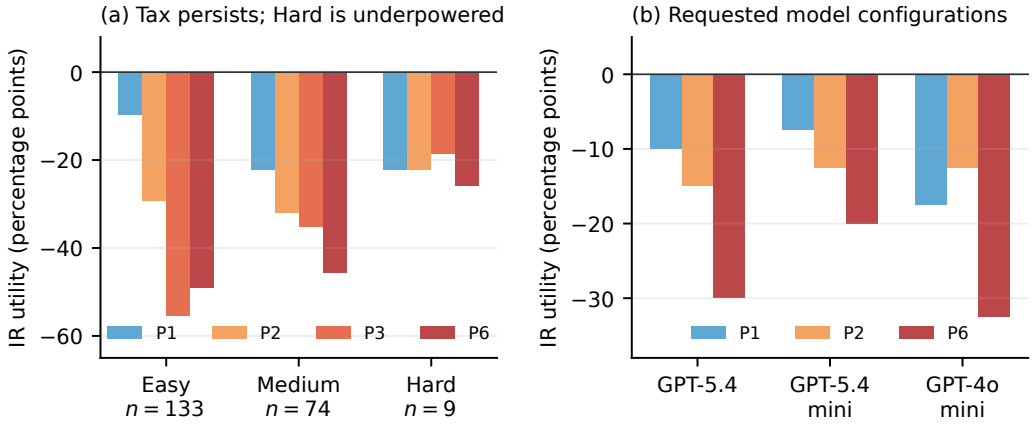


Figure 5: Secondary robustness analyses. (a) Pooled IR utility across the automatically assigned difficulty strata of the three-repeat study. The Hard stratum contains nine tasks and does not establish a trend. (b) Pooled IR utility on the 40-task, one-repeat sensitivity subset. The three requested configurations are labels, not an empirically ordered capability scale.

C.2 Extension Benchmarks

Table 3 reports the absolute pass rates behind Figure 6. Of the 35 relay contrasts, 33 are negative.

A policy other than P0 has the best row value in two cases. P2 leads on ArchXBench Medium with 58.3% against 54.2%, and P1 leads on ArchXBench Hard with 28.6% against 19.0%. These differences correspond to one and two additional passing tasks and have Holm-adjusted $p = 1.0$.

The largest deficits occur where P0 has the most headroom, most clearly on ArchXBench Easy, where P0 passes 96.2%. Because these are one-repeat results, we use them to test direction and possible crossover rather than to rank policies.

Table 3: One-repeat pass rates (%) on the extension benchmarks. Bold marks the best value in each row. The three-way ArchXBench grouping is derived from its ordered native levels, and the two CVDP labels are official for this split.

Benchmark	Tier (n)	P0	P1	P2	P3	P4	P5	P6	P_{hom}
ArchXBench	Easy (26)	**96.2**	65.4	53.8	57.7	26.9	76.9	38.5	42.3
	Medium (24)	54.2	50.0	**58.3**	25.0	12.5	41.7	20.8	45.8
	Hard (21)	19.0	**28.6**	9.5	4.8	4.8	4.8	4.8	0.0
CVDP	Easy (32)	**46.9**	43.8	43.8	9.4	31.2	25.0	25.0	37.5
	Medium (28)	**21.4**	17.9	7.1	3.6	7.1	7.1	3.6	17.9

C.3 RealBench Supplement

Table 4 reports the 10 RealBench modules with three repeats and no RTL debugging. P0 and P1 tie at 43.3%, but P0 uses about 40% fewer tokens per task. Every deeper relay has lower observed correctness and higher cost than P0.

The C++ paths perform worst. P3 and P6 each pass 6.7% of runs, which inflates their tokens per passing solution. With 10 tasks from three IP families, these results extend coverage to modules drawn from larger designs but support only directional conclusions.

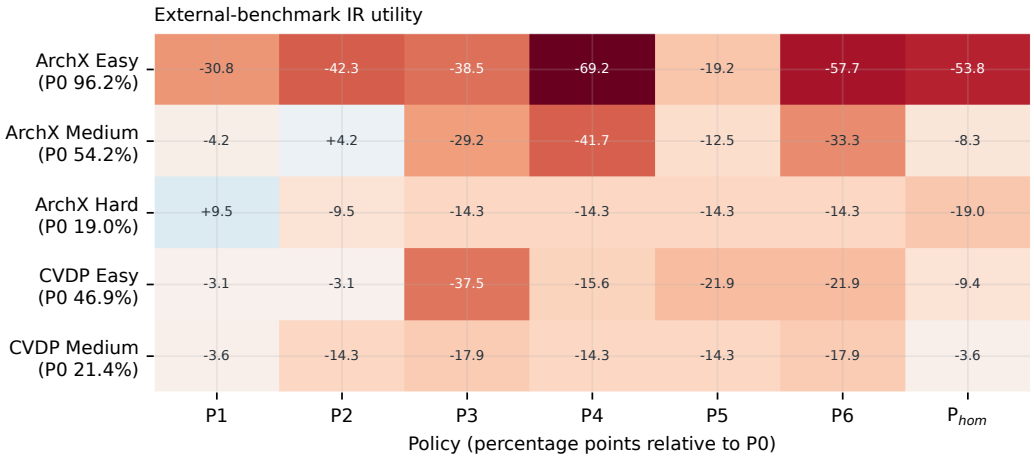


Figure 6: IR utility on the 131-task extension. Values are percentage-point differences from the P0 pass rate printed beside each tier. The two positive cells correspond to one or two additional passes and are not statistically supported.

Table 4: Directional results on the 10-task RealBench subset with three repeats and no RTL debugging. Bold marks the best entry in each metric.

Policy	Pass	Tokens/task	Tokens/pass
P0	**43.3%**	**9,176**	**21,176**
P1	**43.3%**	15,343	35,407
P2	36.7%	23,161	63,167
P3	6.7%	22,261	333,920
P4	13.3%	29,835	223,766
P5	30.0%	28,923	96,411
P6	6.7%	42,222	633,332
P_{hom}	40.0%	29,964	74,911

C.4 Software-IR Tax Area

The fixed-budget tax $T_Q(P_i; B)$ is integrated over $B \in \{16\text{k}, 32\text{k}, 48\text{k}\}$ with the trapezoidal rule. It is then divided by the width of the budget range to give the normalized average $\bar{T}_{\text{AUC}} = T_{\text{AUC}}/(48\text{k} - 16\text{k})$. Larger values indicate a larger deficit relative to P0 across the budget range.

The normalized tax measures the average budgeted-success deficit across the evaluated range. For P1 on VerilogEval, 0.185 corresponds to 18.5 percentage points, close to its 18.4-point unbudgeted pass-rate deficit. Differences from the unbudgeted deficit reflect budget filtering of successful runs. Table 5 reports a positive normalized tax for every relay on both benchmarks.

Table 5: Normalized average Software-IR Tax $\bar{T}_{\text{AUC}}$ over budgets of 16k, 32k, and 48k tokens. Bold marks the smallest tax in each column.

Policy	VerilogEval v2	RTLLM
P1	**0.185**	**0.060**
P2	0.410	0.168
P3	0.567	0.278
P4	0.595	0.337
P5	0.435	0.145
P6	0.689	0.377
P_{hom}	0.468	0.267

C.5 Matched-Rule Control

The matched-rule control asks whether the relay deficit reflects asymmetric final-prompt rules. The three final RTL conditions share a byte-identical hardware-rule prefix and differ only in their source.

- A0 receives `StructuredInfo`.
- A2 receives a frozen Python artifact.
- A3 receives a frozen C++ artifact.

A2 and A3 do not receive the contract. The prespecified primary scope is a 20-task VerilogEval subset, and the mixed 40-task pool is secondary. Holm correction is applied within each family.

Table 6 shows that equal rules do not produce parity. In the primary scope, both relay sources trail A0 by 30 points, although the small sample leaves this imprecise (adjusted $p = 0.094$). In the secondary pool, the deficits are 22.5 and 20.0 points (adjusted $p = 0.023$ for both). Because A2 and A3 still lack the contract, this control addresses rule asymmetry but not contract access. The persistent-contract control instead measures the effect of adding a software view while keeping the authoritative contract available in every condition.

Table 6: Matched-rule relay control with one repeat. Adjusted p is Holm-corrected within the primary or secondary family.

Scope	Condition	Pass	ΔA0	Adjusted p
Primary VE ($n = 20$)	A0 contract	**85.0%**	–	–
	A2 Python	55.0%	−30.0	0.094
	A3 C++	55.0%	−30.0	0.094
Secondary pool ($n = 40$)	A0 contract	**62.5%**	–	–
	A2 Python	40.0%	−22.5	0.023
	A3 C++	42.5%	−20.0	0.023

C.6 Orphan-Reference Ablation

Design. The evaluated C++-to-RTL template contains five rules that consult an original specification, but the main-study renderer supplies no specification to this call (Appendix D). To test whether this defect explains the C++ deficit, we prespecify a benchmark-stratified sample of 60 tasks, with 45 from VerilogEval v2 and 15 from RTLLM. We run three repeats of three interleaved final-lowering conditions.

- P0 re-renders the direct prompt from the frozen `StructuredInfo`.
- A_{orig} reuses the byte-identical saved C++-to-RTL prompt.
- A_{fix} keeps the same C++ artifact and makes the five rules self-contained or removes them.

Every condition makes one final RTL call with the same model and output cap and receives no RTL repair. Only $A_{fix}-A_{orig}$ is a one-factor prompt comparison. P0 keeps the native direct-path system message and serves as a contemporaneous level check. The manifest, templates, call envelope, and rotation were hash-frozen before the run, and all 540 final runs were measured.

Results. At the task level, $A_{fix}-A_{orig}$ is 0.0 points, with a 95% CI of $[-3.9, +3.9]$ and an adjusted CI of $[-4.4, +4.4]$. The win, loss, and tie counts are 6, 6, and 48. The adjusted interval crosses zero and extends beyond $+3$ points, so the contrast is inconclusive at the registered margin. It is not evidence that the edit has no effect.

In contrast, $A_{fix}-P0$ is -45.6 points, with an adjusted CI of $[-57.2, -33.9]$ and win, loss, and tie counts of 0, 38, and 22. The repair therefore neither restores parity nor accounts for the observed shortfall. The remaining translator heuristics, the quality of the frozen C++ artifacts, and the cause of the residual gap stay unresolved.

This experiment shares 22 tasks with the persistent-contract control and is not pooled with it. Its costs cover the final RTL call only.

Table 7: Targeted C++-lowering prompt ablation on 60 tasks with three repeats. Costs cover the final RTL call only, and the frozen upstream C++ construction is excluded. Bold marks the best observed value in each metric.

Condition	Final-call input	Pass	Tokens/call	Tokens/pass
P0	Frozen `StructuredInfo`	**82.8%**	2,573	**3,109**
A_{orig}	Frozen C++, original prompt	37.2%	**1,586**	4,261
A_{fix}	Same C++, orphan references removed	37.2%	1,620	4,352

Table 8: Controlled factors in the persistent-contract control. "Same" means byte-identical within a task. All three conditions use the same task manifest and three-repeat schedule.

Prompt or evaluation factor	PC0	PC2	PC3
System message and common RTL rules	Same	Same	Same
Authoritative `StructuredInfo` contract	Same	Same	Same
Source-precedence paragraph	Same	Same	Same
Optional auxiliary view	None	Frozen Python	Frozen C++
Model, output cap, and no-fallback gate	Same	Same	Same
Evaluator and no-RTL-debug rule	Same	Same	Same

C.7 Persistent-Contract Control

Design. The persistent-contract control estimates the incremental artifact effect defined in Section 3. The protocol, task manifest, benefit margin, and prompts were hash-frozen before any API call. The primary scope contains 100 tasks unused in an earlier pilot, with 75 from VerilogEval v2 and 25 from RTLLM. The frozen views are the Python and C++ artifacts recovered from the final RTL-stage prompts of P2 and P3 in repeat `r0`, following local checks and any triggered repairs. Table 8 lists the controlled factors. Within each task, the three conditions share the system message, the hardware rules, the authoritative contract, and the source-precedence paragraph, and differ only in the optional view. Each condition runs three repeats with GPT-5.4 under strict routing, with condition order rotated within task–repeat blocks. Each final RTL output is evaluated once without repair. Malformed-response retries count toward cost and never receive testbench feedback. The operative template is listed as D9 in Appendix D.

Primary results. Table 9 reports the primary estimates. PC0 passes 79.0%, PC2 78.3%, and PC3 78.7%. The paired task outcomes show why the aggregate differences are near zero. PC2 wins on four tasks, loses on six, and ties on 90. PC3 wins on two, loses on two, and ties on 96. The Bonferroni-adjusted upper endpoint for PC3 is $+2.33$ points. This lies below the $+3$-point margin and rules out a benefit of that size within the prespecified two-contrast family. The endpoint for PC2 is $+3.67$ points, so the same claim remains inconclusive for the Python view. Neither result establishes equivalence. The inference and the operational outcome agree. Neither view improves observed accuracy. Final-call tokens per run rise by 18.8% and 35.6%, and final-call tokens per passing solution rise by 19.9% and 36.2%. These figures exclude the construction of the views, which would only add to the relay cost.

Table 9: Persistent-contract control on 100 primary tasks with three repeats. Every condition receives byte-identical RTL rules, the same authoritative `StructuredInfo` contract, and identical source-precedence instructions, and only the optional frozen view changes. Token columns cover the final RTL calls, including formatting retries and excluding view construction. Mean costs are per task–repeat run, and tokens per passing solution include spending on failed runs. Intervals use task-level paired bootstrap resampling. "Adj. CI" is a 97.5% two-sided interval per contrast, a Bonferroni adjustment targeting nominal 95% simultaneous coverage across the two contrasts. Bold marks the best observed value in each performance metric.

Condition	Optional view	Pass	ΔPC0	95% CI	Adj. CI	Final-call tokens/run	Final-call tokens/pass
PC0	None	**79.0%**	–	–	–	**1,425.6**	**1,804.5**
PC2	Frozen Python	78.3%	-0.67	$[-4.33, +3.00]$	$[-5.00, +3.67]$	1,694.3	2,162.9
PC3	Frozen C++	78.7%	-0.33	$[-3.00, +2.00]$	$[-3.67, +2.33]$	1,933.2	2,457.5

Hard supplement. The 15 ArchXBench Hard tasks are reported separately because they are outside the primary estimand. PC0 and PC2 both pass 13.3%, and PC3 passes 4.4%. The paired differences are 0.0 points (95% CI $[-13.3, +11.1]$) for PC2 and -8.9 points ($[-20.0, 0.0]$) for PC3. With 15 tasks and accuracy near the floor, these estimates show no observed reversal on hard tasks and do not establish a ranking of the two languages.

Pilot. An earlier 40-task, one-repeat pilot motivated this higher-powered design and is not pooled with it. Its larger negative point estimates did not reproduce. On 100 new tasks with three repeats, both contrasts lie within one point of zero. We treat the confirmatory control as the primary result and the pilot only as design history.

Integrity audits. All 17 integrity audits pass. All 1,035 final runs were measured with no fallback model, cap deviation, current-run truncation, or testbench leakage.

A gateway outage produced 117 regeneration events covering 116 unique runs, all before any model response existed. One PC0 run was attempted twice. The numbers of unique regenerated runs are 37, 37, and 42 for PC0, PC2, and PC3.

Excluding all 17 affected primary tasks yields pass rates of 79.52%, 78.71%, and 79.12%, with differences of -0.80 and -0.40 points. The outage therefore does not drive the conclusion.

Table 10: Prompt and information protocol for the evaluated policies and controls. The executor column distinguishes model calls from external checking and evaluation.

Stage	Executor	Stage input	Output / constraint
Understanding	LLM	Raw specification and multimodal context; shared preprocessing	Frozen `StructuredInfo`
Test-case extraction	LLM	Raw specification; shared preprocessing	Frozen spec-derived tests
Direct RTL generation (P0)	LLM	`StructuredInfo`	Raw synthesizable Verilog only
Pseudocode construction	LLM	`StructuredInfo`	JSON plan; no code outside the schema
Plan → RTL	LLM	Module name, classification, ports, state elements, logic steps, and clock/reset information	Raw synthesizable Verilog only
Plan → Python	LLM	Plan logic steps only	JSON-wrapped Python model
Plan → C++	LLM	Module name, classification, ports, state elements, and logic steps	JSON-wrapped C++ model
Python → RTL	LLM	Module name and Python artifact; Python limited to the first 6,000 characters	Raw synthesizable Verilog only
C++ → RTL	LLM	C++ artifact only; optional specification and self-test-status blocks inactive	Raw synthesizable Verilog only
Matched-rule lowering	LLM	One source per condition: `StructuredInfo`, Python, or C++	Identical final RTL rules; Python/C++ conditions omit the contract
Persistent-contract lowering	LLM	Identical `StructuredInfo`, with an optional frozen Python or C++ view	Contract authoritative; identical rules and source precedence
Executable-IR validation	Local checker	Generated IR and frozen spec-derived tests	Pass/fail and failure observations, including expected/observed values when available
Blind IR repair	LLM	Bounded prefix of the current IR and bounded failure report	Revised IR; at most two repair calls per checked stage
Final evaluation	Benchmark harness	Generated RTL and official testbench	One functional verdict per generated RTL; no LLM-driven RTL debugging or candidate selection

D Prompt and Information Protocol

D.1 Information Flow

All main-study policies use the same frozen model and sampling protocol. This means GPT-5.4 at low reasoning effort, no fallback model, three independent repeats, and a downstream budget grid of 16k, 32k, and 48k tokens. Prompt templates and prompt builders are versioned in the released implementation, and every rendered prompt is retained and hash-logged with its run.

The shared understanding stage produces the frozen `StructuredInfo` contract. Every policy starts from this object, and no downstream call receives the original specification text.

Table 10 distinguishes LLM calls from external validation and evaluation. *Stage input* lists the data consumed by the executor named in each row; only rows with an LLM executor describe model-visible inputs. The raw specification is available only during shared preprocessing. Downstream LLM calls receive neither the raw specification nor the full frozen test suite or official testbench. Executable-IR validation runs locally, and blind repair receives only a bounded IR prefix and a bounded failure report. The two fairness controls replace the final prompt envelope as shown while retaining the same evaluator.

An audit of the rendered prompts found that the 6,000-character guard on Python-to-RTL payloads clipped 2 of 1,850 payloads. Both come from two repeats of the same RTLLM asynchronous-FIFO task. The third repeat stayed below the guard and also failed under the same policy, but this does not identify the missing counterfactual, so we report the guard as a limitation. The 2,500-character bound on the IR prefix applies only to blind-repair prompts, after an executable IR has already failed its local check.

D.2 Prompt Listings

Listing policy. The symbols D0 to D9 are manuscript labels for prompt stages, not identifiers in the implementation. They map to the policies as follows.

- P0 uses D0.
- P1 uses D1 and D2.
- P2 and P4 use D3 and D4.
- P3 and P5 use D5 and D6.
- P6 composes D1, D3b, D7, and D6.
- P_{hom} inserts D8 between D3a and D4.
- The persistent-contract control uses D9.
- A1 and A2 are the shared test-extraction and blind-repair prompts.

Because transition-specific payloads are part of the implemented policies, we show the operative excerpts needed to reconstruct each information boundary. Angle-bracketed text denotes a task-dependent payload, and braces mark an active conditional branch. Inactive branches and worked examples are omitted consistently. The full rendered prompts and hashes in the supplementary artifact are authoritative. The excerpts preserve every model-visible field and every rule material to the comparison while shortening examples and repeated boilerplate.

D0. `StructuredInfo` → RTL (P0, operative excerpt).

```
You are a senior RTL design engineer. Generate synthesizable
Verilog code for the following hardware module.

## Structured Module Information
<STRUCTURED_INFO_JSON>

## Requirements (Strict)
1. Completeness -- Output a complete Verilog module with all ports,
   internal wires/regs.
2. Bit Ordering -- If the spec gives explicit bit-index examples,
   follow them exactly. Do NOT invent your own ordering.
3. Combinational vs Sequential:
   - If combinational: use `assign` or `always @(*)`, no clock/reset.
   - If sequential: use clocked `always @(posedge clk ...)` and reset.
4. State Elements -- Declare `reg` for each state element; initialise
   in reset.
5. Coding Style -- 2-space indent, comments, no simulation-only code.
6. Cross-Module Instantiation -- If sub-modules are needed, use exact
   names and port mappings from `previous_rtl_ports`.
7. Output Format -- Output ONLY raw Verilog code. Do not use Markdown
   fences or explanation. Start with `module`, end with `endmodule`.

<WORKED EXAMPLE OMITTED; RETAINED IN THE ARCHIVED PROMPT>
```

```
Now generate the Verilog RTL for the given module.
```

D1. `StructuredInfo` → pseudocode (operative excerpt).

```
You are a Senior Hardware Architect. Convert the given sub-module
specification into a rigid architectural plan in pseudocode.

## Input Specification (JSON):
<STRUCTURED_INFO_JSON>

## Task
1. Determine hardware classification based on `state_elements` and
   `functionality`:
   - Empty state_elements -> COMBINATIONAL
   - Sequence of stages/pipeline -> SEQUENTIAL_PIPELINE
   - Explicit states/transitions -> STATE_MACHINE
   - Default to SEQUENTIAL_PIPELINE if state exists but is ambiguous.
2. Define exact I/O from `inputs` and `outputs`.
   - `width` MUST be a string: "8", "128", or "1", never an integer.
   - Include `direction`: "input" or "output".
3. Copy state elements when present; otherwise use an empty list.
4. Write clear step-by-step logic. For sequential logic, describe
   each clock edge and reset behavior.
5. Clock and reset:
   - For combinational designs, use null.
   - For sequential designs, use the supplied configuration or infer:
     clk, rst_n, ACTIVE_LOW, POSEDGE.

## Output Format
Respond with ONLY valid JSON. No markdown, extra text, or trailing comma.

### Required Schema
{
  "module_name": "string (sub_module_name from input)",
  "hardware_classification":
    "COMBINATIONAL | SEQUENTIAL_PIPELINE | STATE_MACHINE",
  "inputs_outputs": {
    "signal_name": {
      "direction": "input | output",
      "width": "string"
    }
  },
  "state_elements": ["string"],
  "logic_steps": "string (step-by-step pseudocode)",
  "clock_reset_config": null | {
    "clock_signal": "string",
    "reset_signal": "string",
    "reset_polarity": "ACTIVE_LOW | ACTIVE_HIGH",
    "clock_edge": "POSEDGE | NEGEDGE"
  }
}

Example 1: Combinational (constant zero)
{
  "module_name": "TopModule",
  "hardware_classification": "COMBINATIONAL",
  "inputs_outputs": {
    "zero": {"direction": "output", "width": "1"}
  },
  "state_elements": [],
  "logic_steps": "Assign constant 0 to output 'zero'.",
  "clock_reset_config": null
}

Example 2: Sequential with pipeline
{
  "module_name": "SimpleAdder",
  "hardware_classification": "SEQUENTIAL_PIPELINE",
  "inputs_outputs": {
    "clk": {"direction": "input", "width": "1"},
    "rst_n": {"direction": "input", "width": "1"},
    "a": {"direction": "input", "width": "8"},
    "b": {"direction": "input", "width": "8"},
    "sum": {"direction": "output", "width": "9"}
  },
  "state_elements": ["sum_reg"],
  "logic_steps": "On positive clk: if (!rst_n) sum_reg <= 0;
    else sum_reg <= a + b; assign sum = sum_reg;",
  "clock_reset_config": {
    "clock_signal": "clk", "reset_signal": "rst_n",
    "reset_polarity": "ACTIVE_LOW", "clock_edge": "POSEDGE"
  }
}

Now generate the pseudocode plan for the given sub-module.
```

D2. Pseudocode → RTL (operative excerpt).

```
You are a senior RTL design engineer. Generate synthesizable
Verilog code for the following hardware module.

## Module Specification
- Name/class: <MODULE_NAME> / <CLASSIFICATION>
- Ports (JSON): <PORT_JSON>
- State Elements: <STATE_ELEMENTS>
- Pseudocode Logic Steps:
<LOGIC_STEPS>

{Clock/reset branch: exact clock, reset, polarity, and synchronous
style; or "Combinational Module: No clock or reset."}

{Optional Previously Generated Modules port-list branch}

## Requirements (Strict)
1. Completeness -- Output a complete module with all ports and
   internal wires/regs.
2. Combinational vs Sequential:
   - COMBINATIONAL: `always @(*)`, blocking `=`, no clock/reset.
   - Otherwise: clocked `always @(posedge <CLOCK> ...)` with reset.
3. State Elements -- Declare each as `reg`; initialise in reset.
4. Coding Style -- 2-space indent, comments, no simulation-only code.
5. Cross-Module Instantiation -- Use exact module names and port
   mappings from `previous_rtl_ports`.
6. Output Format -- ONLY raw Verilog, no Markdown or explanation.
   Start with `module` and end with `endmodule`.

<WORKED EXAMPLE OMITTED; RETAINED IN THE ARCHIVED PROMPT>

Now generate the Verilog RTL for the given module.
```

D3a. `StructuredInfo` → Python (operative excerpt).

```
SYSTEM: Output ONLY valid JSON.
USER: Generate a bit-accurate executable Python hardware model
from this StructuredInfoDict. Do not use the original spec.
Preserve exact module behavior, ports, bit widths, reset behavior,
and sequential state from the structured information only.

## StructuredInfoDict
<STRUCTURED_INFO_JSON>

## Output -- ONLY valid JSON:
{"python_code": "string with escaped newlines", "test_cases": []}
The Python code should define one class for the module. Use
tick(...) for sequential modules when needed, getters or attributes
for outputs, and mask arithmetic results to hardware bit widths.
```

D3b. Pseudocode → Python (operative excerpt).

```
You are a hardware/software engineer. Translate the pseudocode below
into executable Python code.

## Pseudocode (Hardware Architectural Plan)
<PSEUDOCODE_LOGIC_STEPS>

## Rules
1. Implement as a Python class with the module name as class name
2. Include a `self_test()` static method with test cases
3. Use exact bit-widths where specified
4. For sequential logic, use `tick()` to update internal state
5. Output ONLY valid JSON, no markdown

## Output Format
{
  "python_code": "complete class; escape newlines",
  "test_cases": [
    {"inputs": {"name": value}, "expected": {"name": value}}
  ]
}

Example:
```

```
{
  "python_code": "class TopModule:\n    def __init__(self):\n
    self.state = 0\n    def tick(self, a, b):\n
    self.state = a & b\n    def get_out(self):\n
    return self.state\n    @staticmethod\n    def self_test():\n
    m = TopModule()\n    m.tick(1, 0)\n
    assert m.get_out() == 0\n    return True",
  "test_cases": [
    {"inputs": {"a": 1, "b": 0}, "expected": {"out": 0}}
  ]
}
```

D4. Python → RTL (operative excerpt).

```
SYSTEM: Output ONLY raw Verilog.
USER: Translate this Python hardware model to synthesizable
Verilog.

CRITICAL: The Verilog module MUST be named `<MODULE_NAME>` exactly.

## Python Code
<PYTHON_CODE; AT MOST FIRST 6000 CHARACTERS>

Output ONLY raw Verilog starting with module, ending with endmodule.
```

D5a. `StructuredInfo` → C++ (operative excerpt).

```
You are an expert hardware engineer. Translate the given hardware
specification into synthesizable C++ code.

## Input: Structured Hardware Specification
<STRUCTURED_INFO_JSON>

## Critical Rules

### 1. Hardware Classification
{Branch: COMBINATIONAL means no clock/reset/state and pure functions;
SEQUENTIAL means clock/reset and class/struct state registers.}

### 2. Bit-Accurate Types
Use uint8_t, uint16_t, uint32_t, uint64_t, int8_t, etc. from
<cstdint>. NEVER use plain int or long; widths must match the spec.

### 3. Signed vs Unsigned (CRITICAL)
For signed comparisons, cast operands to int32_t or int64_t first.
Never mix signed and unsigned without explicit casts.

### 4. State Encapsulation
Encapsulate registers, counters, and state in one C++ class.
- Constructor initializes all state to reset values.
- `void tick(inputs...)` updates state and sets outputs.
- Getters or a struct expose outputs.

### 5. Universal HLS Constraints
- No dynamic memory, pointers, recursion, or virtual functions.
- All loop bounds must be statically known.
- No std::cout or I/O in core logic (self_test only).

### 6. Self-Test Function
Include `static bool self_test()` that checks key input/output pairs
and returns true only if every check passes.

## Output Format -- STRICT JSON
Respond with ONLY valid JSON. No markdown or explanations.
{
  "cpp_code": "complete C++ with <cstdint>, class, tick(), and
    self_test(); escape newlines",
  "compiler_directives": "None or comma-separated pragmas",
  "test_cases": [
    {
      "description": "what this test checks",
      "inputs": {"signal_name": value},
      "expected_outputs": {"signal_name": value}
    }
  ]
}

### Validation Rules
- cpp_code must be valid C++ with escaped newlines.
- Supply at least three tests, each with all input and output ports.
- Use integer signal values; use decimal for multi-bit values.

### Example (1-bit constant-zero module)
```

```
{
  "cpp_code": "#include <cstdint>\nclass TopModule {\npublic:\n
    uint8_t compute_zero() { return 0; }\n
    static bool self_test() {\n        TopModule m;\n
        if (m.compute_zero() != 0) return false;\n
        return true;\n    }\n};",
  "compiler_directives": "None",
  "test_cases": [
    {"description": "always outputs 0", "inputs": {},
     "expected_outputs": {"zero": 0}}
  ]
}

Now generate the C++ implementation and test cases for the module.
```

D5b. Pseudocode → C++ (operative excerpt).

```
SYSTEM: Output ONLY valid JSON.
USER: Translate this pseudocode hardware plan into synthesizable,
bit-accurate C++ code. Use only the pseudocode plan below. Do not
use the original spec. Preserve module name, ports, bit widths,
state, clock/reset behavior, and logic steps exactly.

## Module
<MODULE_NAME>
## Hardware Classification
<CLASSIFICATION>
## Inputs/Outputs
<PORT_JSON>
## State Elements
<STATE_ELEMENTS>
## Logic Steps
<LOGIC_STEPS>

## Output -- ONLY valid JSON:
{"cpp_code": "string with escaped newlines",
 "compiler_directives": "None"}
Define one class/struct, use fixed-width integer types, avoid
dynamic memory, and include self_test() when possible.
```

D6. C++ → RTL (abridged evaluated template).

```
You are a senior RTL design engineer. Translate the following C++
hardware model into synthesizable Verilog RTL.
## C++ Source Code (the Intermediate Representation)
<CPP_CODE>
## CRITICAL -- Hardware Semantics NOT Expressible in C++
The C++ may be correct as software, but hardware requires semantics
that C++ cannot express. You MUST infer or add these:
### 1. Combinational vs Sequential
- COMBINATIONAL: use `assign` or `always @(*)`; no clock/reset.
  Ignore class state if the original design is combinational.
- SEQUENTIAL: use `always @(posedge clk)` with synchronous reset
  unless the specification explicitly says asynchronous.
### 2. Bit-Widths
- uint8_t -> [7:0], uint16_t -> [15:0], bool -> one bit.
- If the specification and C++ disagree, the SPEC wins.
### 3. Reset Initialization
- Map C++ constructor values to explicit reset logic.
- Use reset logic, not only simulation-time `initial` blocks.
- Give every register, counter, and state element a reset value.
### 4. Signed Operations
- Map C++ signed arithmetic to Verilog `$signed()`.
- Example: `(int32_t)a < (int32_t)b` becomes
  `$signed(a) < $signed(b)`.
### 5. Loop Unrolling
- Fixed-range C++ for loops become generate loops or unrolled assign.
- Fixed-iteration while loops receive the same treatment.
### 6. Output Timing
- A same-tick C++ output is combinational (Mealy).
- A next-tick output is registered (Moore).
- Match the specification's timing, not necessarily the C++ timing.
## Requirements
1. Output a COMPLETE Verilog module with header, ports, internal
   wires/regs, logic, and endmodule.
2. Output raw Verilog beginning with `module`; no Markdown fences.
3. Follow exact Verilog rules in the original specification.
4. Use 2-space indent and clear signal names.
```

```
5. Give every output register a reset value.
## Output Format
Output ONLY raw Verilog. Start with `module`, end with `endmodule`.
```

Prompt-audit note. The renderer supplied the C++ artifact to this call but neither the raw specification nor an activated self-test-status block. Those optional template branches therefore did not appear in evaluated calls, and the remaining references to a "specification" had no corresponding payload. Five rules are affected, including the rule on asynchronous reset, the rule that the specification wins on widths, the rule on output timing, and the requirement to follow Verilog rules in the original specification. We report the evaluated translator rather than replace it retroactively, and Appendix C.6 tests a local repair while freezing the C++ artifact. The signed-cast and same-tick or next-tick heuristics are not generally semantics-preserving. Results on C++ paths therefore characterize the implemented translator rather than an intrinsic limitation of C++ intermediates. D9 separately tests lowering with an authoritative contract.

D7 and D8. Depth controls (operative excerpts).

```
D7 SYSTEM: Output ONLY valid JSON.
D7 USER: Translate this Python hardware model to synthesizable C++.
## Python Code
<PYTHON_CODE>
## Rules
Translate the class with the same logic; use fixed-width integer
types; initialize all state; keep tick() and self_test(); include
<cstdint>; output ONLY valid JSON containing cpp_code and
compiler_directives.

D8 SYSTEM: Perform behavior-preserving Python reserialization only.
D8 USER: Reserialize the following Python hardware model while
preserving its exact behavior, module interface, port names, bit
widths, and clock/reset semantics. Do not add, delete, or reinterpret
functionality. Return ONLY valid JSON with one field named
python_code.
<PYTHON_CODE; AT MOST FIRST 6000 CHARACTERS>
```

A1. Shared test extraction.

```
A1 SYSTEM: Output ONLY valid JSON.
A1 USER: Extract test cases from this hardware spec. For each,
specify ALL input port values and expected output values as integers.
## Spec: <ORIGINAL_SPEC; FIRST 4000 CHARACTERS>
## Output -- ONLY valid JSON:
{"test_cases": [{"inputs": {"port": val}, "expected": {"port": val}}]}
Include at least 3 test cases covering normal and edge cases.
```

A2. Blind IR repair.

```
A2 SYSTEM: Output ONLY valid JSON.
A2 USER: Fix this <Python|C++> code that FAILED its test.
## Error: <FAILURE_LOG; FIRST 800 CHARACTERS>
## Code: <CURRENT_IR; FIRST 2500 CHARACTERS>
Output valid JSON with fixed <python_code|cpp_code>.
```

D9. Persistent-contract final lowering (operative template).

```
SYSTEM: You are a senior RTL design engineer expert in Verilog.
USER: You are a senior RTL design engineer. Produce a synthesizable
Verilog implementation of the hardware described below.

## Hardware rules (identical in PC0/PC2/PC3)
1. MODULE AND PORTS. Name the module `<MODULE_NAME>` exactly. Match
   every port name, direction, and width. Do not add or drop a port.
2. COMBINATIONAL VS SEQUENTIAL. Decide from the description.
   Combinational logic uses `assign` or `always @(*)` and has no
   clock/reset. Sequential logic uses `always @(posedge clk)`.
3. STATE UPDATES. Use non-blocking `<=` inside clocked blocks and
   blocking `=` inside combinational blocks.
4. RESET. Implement explicit reset logic, not `initial` blocks alone.
```

```
   Use synchronous reset unless asynchronous reset is specified; then
   include it in the sensitivity list. Match polarity and values, and
   give every register, counter, and state element a reset value.
5. WIDTHS AND SIGNEDNESS. Match widths exactly. Use `$signed()` and
   `$unsigned()` where required. Truncate or extend explicitly.
6. TIMING. Honour clock period, output latency, and state holding.
   Register an output when it is specified to hold its value.
7. SYNTHESIZABILITY. Avoid latches, multiple drivers, and
   non-synthesizable constructs. Give combinational signals defaults.
8. OUTPUT FORM. Output ONLY the synthesizable Verilog module.
9. NO EXTRAS. No Markdown, testbench, explanation, or commentary.
   Start directly with `module` and end with `endmodule`.

## Authoritative hardware contract
The StructuredInfo below is the authoritative hardware contract for
this module.
<STRUCTURED_INFO_JSON; BYTE-IDENTICAL ACROSS CONDITIONS>

## Optional auxiliary software view
PC0 body: NONE
PC2 body: The frozen Python behavioral model of the same module:
```python
<FROZEN_PYTHON_ARTIFACT>
```
PC3 body: The frozen C++ behavioral model of the same module:
```cpp
<FROZEN_CPP_ARTIFACT>
```

Source precedence
The StructuredInfo contract is authoritative. The optional software
artifact is only an auxiliary behavioral view. If they disagree or
the auxiliary artifact omits a hardware detail, follow StructuredInfo
and the common hardware rules.
```

Reading the listings. No main-study downstream call receives the raw specification, but the payload depends on the transition, as Table 10 shows.

- D3b receives only the logic steps of the plan, and D5b receives typed plan fields.
- D4 receives the module name and the Python artifact, with no port list, clock and reset configuration, or hardware rules.
- D6 receives only the C++ artifact.

These choices define the complete-policy estimand and should not be read as a comparison of representations alone. D9, in contrast, supplies the same frozen `StructuredInfo` to all three control conditions, so its only varying block is the optional software view.

The only raw-specification call among the listings is A1, a shared preprocessing step that freezes test vectors before any policy runs. These vectors are executed locally and never copied into generation prompts. When repair is triggered, A2 exposes only a bounded failure log and a bounded prefix of the current IR. Repair is limited to two rounds and never sees the official testbench. This separation is what allows the self-test results in Appendix B to be read as evidence about the local success signal rather than about leakage from the evaluator.