# AFSIM-AGENT: HIERARCHICAL SELF-REFLECTIVE CODING AGENT FOR CROSS-LANGUAGE ADAPTATION TO UNSEEN SCRIPTING LANGUAGES

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Code agents solve many tasks, but their competence rests on pretraining that supplies the language's syntax and abundant examples. Domain-specific languages (DSLs) break both: models lack the domain's grammar, cannot interpret compiler diagnostics, and have few examples to learn from. We study AFSIM, an open-source multi-domain modeling and simulation framework from the US Air Force Research Laboratory (AFRL), whose scripting language is rare in training data. General coding agents fail often when writing it; we aim for an agent that masters an unfamiliar DSL without retraining. We propose **AFSIM-Agent**, with five modules. (1) A **curated domain knowledge surface** turns the DSL into retrievable priors: a read-only document tree of grammar, documentation, and worked examples, plus skill cards linking engineering tasks to source files. (2) An **executable feedback ladder** supplies four levels of signal—static constraints, minimal-harness compilation, full simulation, and independent fidelity audit—constructing the trace-level signals general agents never observe. (3) A **verification-gated repair loop** admits only edits verified at the current file revision, so each round builds on verified state. (4) A **locally in-distribution (LID) context discipline** keeps every LLM call in-distribution and the main context limited to decomposition strategy, control flow, and normalized state, so long-horizon tasks do not dilute critical signals. (5) A **deterministic plan-audit gate and anti-reward-hacking guardrails** constrain the plan before code is written and the result against the specification at submission. We also build **AFSIM-Bench**: 114 tasks with compilable reference code and a reproducible measurement instrument for this domain. On the 34 executable tasks, the bare model passes 17.6%, the three general harnesses reach 41.2%–47.1%, and AFSIM-Agent reaches 58.8%; the harness buys the ability to make a project run, and domain grounding buys the ability to make it satisfy intent.

## 1 INTRODUCTION

Code agents succeed on general-purpose languages because syntax is in-distribution, compilers and tests provide cheap local signals, and examples abound. We study **AFSIM**, an executable modeling-and-simulation DSL from the US Air Force Research Laboratory (AFRL), where all three assumptions weaken: its vocabulary is rare in public code, diagnostics are unstructured, and similarly structured example projects are scarce. More importantly, correctness depends on events in a simulation trace, not on compilation or a clean exit. A project may run to completion without producing the required events or interactions; we call this gap between "runs" and "satisfies intent" the central problem.

General coding agents usually address such tasks with a small set of recurring methods: tool-using ReAct-style loops, repository search and context retrieval, planner–executor decomposition, iterative edit–test–repair, and multi-agent review or critique (Yang et al., 2024; Xia et al., 2025). These methods are effective when the language is in-distribution and compiler or test feedback is aligned with task intent, but they do not supply the missing DSL grammar or decide trace-level behavior.



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.

For executable AFSIM tasks, intent is a predicate over the simulation event trace rather than over source text. We therefore distinguish structure, runtime, behavior, and fidelity checks: a project may be well formed, run cleanly, and still never produce the required events at the required time.

**Contributions.** Our contributions make this failure mode measurable, correctable, and reproducible.

- **C1: Problem definition.** We formalize intent over simulation traces and summarize six design requirements for low-resource executable DSLs. This turns "the code ran" into a testable behavioral claim and exposes why compiler-only feedback is insufficient.
- **C2: AFSIM-Agent.** We build a domain-grounded agent with curated knowledge, executable feedback, revision-gated repair, context discipline, and anti-bypass guards (§2). The benefit is an agent that can adapt to an unseen DSL without retraining and continue repairing toward behavior-level intent instead of stopping at a runnable project.
- **C3: AFSIM-Bench.** We release 114 measured tasks, including 34 executable tasks scored at the behavior level rather than by compilation or exit status (§3). This provides a reproducible instrument that separates harness capability from domain grounding. On the 34 executable tasks, the bare model passes 17.6%, three general harnesses reach 41.2%–47.1%, and AFSIM-Agent reaches 58.8%.

AFSIM-Agent sits at the intersection of code-agent harnesses, software-engineering benchmarks, and domain-specific code generation. Its novelty is not any single harness mechanism, but the combination of domain grounding, executable feedback, and behavior-level measurement for a low-resource executable DSL.

## 2 AFSIM-Agent: A Coding Agent Grounded in Domain Knowledge

### 2.1 System overview and the agent loop

AFSIM-Agent is a hierarchical, self-reflective coding agent that adapts to an unfamiliar DSL inside the harness, without retraining. The control loop has three phases: `base` handles research and routing; `plan` forms a build plan in a read-only environment and passes a deterministic plan audit; `build` generates and repairs the project under write permission, continuously receiving write-time compile probes and full-simulation feedback. The plan-to-build transition is enforced by code rather than prompt convention, and compile feedback during build enters the loop through automatic probes, so verification becomes part of the generation process.

Five cooperating modules support this loop: a curated domain knowledge surface, an executable feedback ladder, a verification-gated repair loop, an LID context discipline, and fidelity, honesty, and anti-bypass guardrails. Figure 1 gives an overview.

The agent implements five modules, shown in Figure 1 and summarized below.

- **Curated domain knowledge surface.** A read-only document tree, keyword index, and skill cards convert grammar, examples, and engineering recipes into retrievable DSL priors without vector search.
- **Executable feedback ladder.** Static checks, write-time compile probes, full simulation, and post-run audits expose increasingly intent-aligned signals, from local syntax errors to event-trace verdicts.
- **Verification-gated repair.** Every verification result is bound to the current project revision; file mutation invalidates stale evidence, and repair continues only against a re-verified state.
- **LID context discipline.** Long tool outputs are normalized or spilled to disk while runtime, behavior, and fidelity verdicts remain in the main context, so compression cannot erase the signal that directs repair.
- **Fidelity, honesty, and anti-bypass guards.** Plan audit, numeric fidelity checks, capability audit, event-only evidence, and a post-verification edit guard make low-cost shortcuts harder and partial verification explicit.

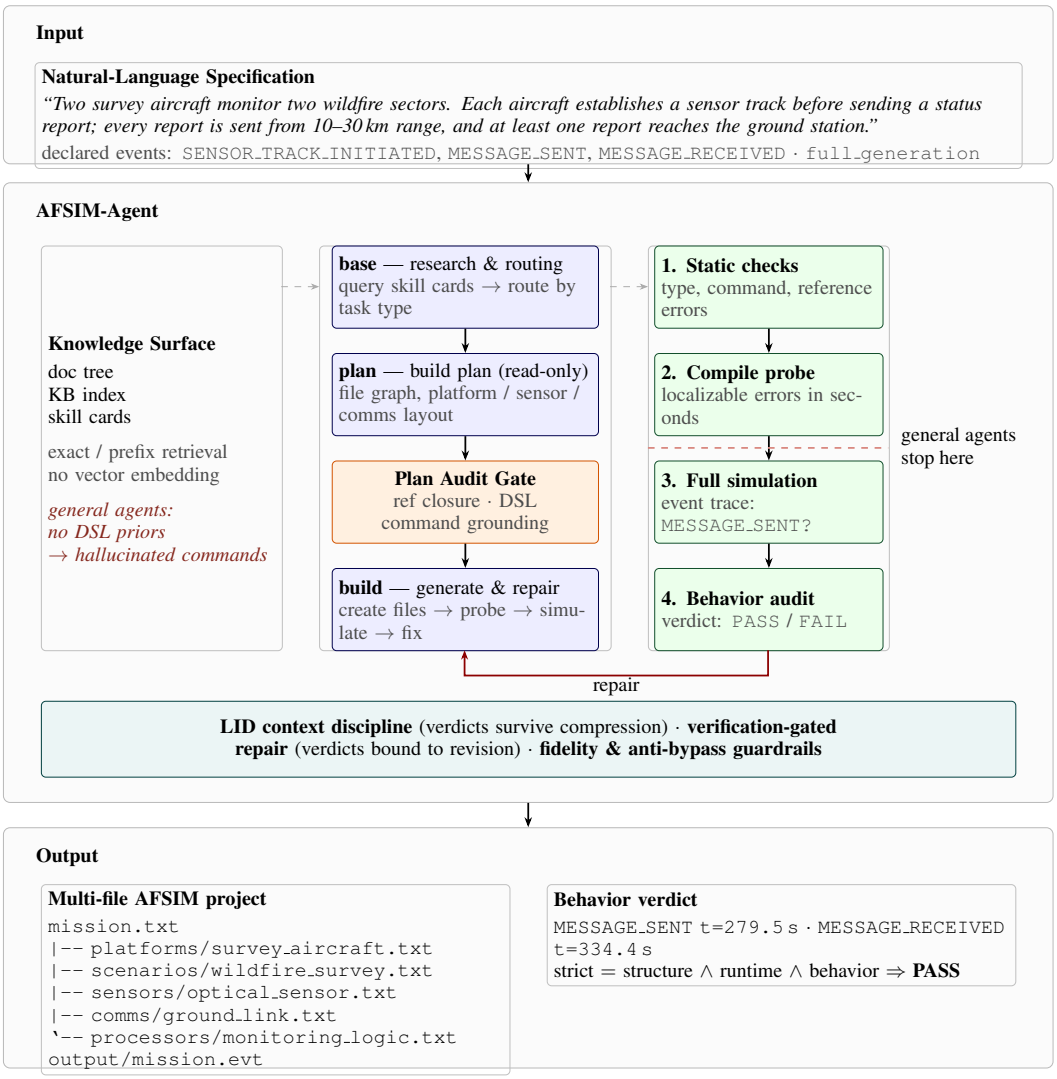


Figure 1: One NL-to-Code run. Input: a natural-language AFSIM task. AFSIM-Agent: knowledge lookup, the plan/build loop, and the four-rung feedback ladder. Output: the generated project and behavior verdict. General agents stop after compilation and execution. **Our run sends a report at 279.5 s and receives it at 334.4 s.**

AFSIM-Agent does not claim to out-design general-purpose agents. Its contribution is to add domain priors and executable, behavior-level feedback to a general workflow; the agent and benchmark are evaluated in three tiers so that harness ability is not credited to domain grounding, or vice versa.

The modules operate as one loop. The knowledge surface grounds the plan, executable feedback observes the build, repair invalidates stale evidence, the context discipline preserves decisive verdicts, and the guards check claims against artifacts. Verification is therefore part of generation, not a post hoc filter: each action either produces new executable evidence or is discarded when that evidence becomes stale. This design also fixes the experiment unit: the task, prompt, scorer, and backbone are held constant while only the surrounding harness or domain grounding changes.

The design is motivated by explicit failure modes rather than by a generic agent pattern. Missing priors cause hallucinated commands; missing behavior observation causes premature completion; context compression can erase evidence needed for repair; and unconstrained completion invites fabricated or out-of-scope evidence. Each module addresses one of these failure modes, which is also what the ablation is designed to test.

The architecture is therefore not a collection of independent prompts. Its modules share the same revision state: knowledge retrieval informs the plan, executable probes update the verification state, repair acts only on that state, and the completion guard rechecks it before submission.

The module boundaries are fixed, but their information flow is shared.

# 3 AFSIM-BENCH: AN INSTRUMENT FOR BEHAVIOR-LEVEL MEASUREMENT

AFSIM-Bench pairs a frozen dataset with executable oracles. The core contains 114 tasks from 25 case families across six types. Its 80 objective tasks test QA, multiple choice, code reading, and fill-in-the-blank; its 34 executable tasks contain 13 bug fixes and 21 full-generation scenarios. Table 1 summarizes this composition.

Table 1: AFSIM-Bench composition.

| Component | Count | Notes |
|---|---|---|
| Objective tasks | 80 | 20 QA, 20 choice, 20 code reading, 20 fill-in-the-blank |
| Executable tasks | 34 | 13 bug-fix, 21 full-generation |
| **Total** | **114** | All gold/frozen |

Executable tasks use four layers. **Structure** checks the declared artifacts; **runtime** requires a clean engine run; **behavior** evaluates event existence, order, counts, forbidden events, and time windows over the normalized trace; and the **strict verdict** is their conjunction. A behavior failure is reported even when an artifact is missing, so the result distinguishes "does not build or run" from "runs but violates intent." Objective tasks are scored deterministically, and executable tasks never execute task-supplied scoring code.

The split between 80 objective and 34 executable tasks is intentional. The objective track measures whether a system can read and manipulate AFSIM artifacts; the executable track measures whether a generated multi-file project behaves as specified. For full generation, the reference project is withheld; for bug fixing, the input is a mutated project. This makes the executable track the test of the central claim rather than another code-reading benchmark.

The public prompt states the required artifacts and event families. Exact thresholds, time windows, event counts, and the reference solution remain hidden. Behavior evidence is accepted only from engine-written event files, and reference solutions plus sabotaged variants validate every executable oracle.

This structure gives the experiments a bounded subject: a system either produces the declared artifacts, runs the entry point cleanly, and satisfies the trace predicates, or it fails at a named layer. The objective track is retained to separate language understanding from executable behavior, not to redefine the paper's headline success metric.

Both tracks use the same frozen evaluator boundary: task inputs and public artifact contracts are visible, while scoring thresholds and reference solutions remain hidden.

# 4 EXPERIMENTS

This section reports two experiments: a horizontal comparison of three tiers on identical tasks — bare model, general harnesses, AFSIM-Agent (§4.2) — and a mechanism ablation of AFSIM-Agent (§4.3). Cost, difficulty calibration and benchmark validity lie outside its scope.

## 4.1 SETUP

Evaluation uses the 114 tasks of §3: 80 objective, 13 `bug_fix`, 21 `full_generation`. Objective tasks are scored by deterministic rules, executable tasks by the frozen behavior-level oracle. No cross-type weighted total is used as a headline metric, because the slices measure different capabilities.

All experiments use one base model, DeepSeek v4.1 Flash (max setting), via API. Nothing is fine-tuned and no retrieval index is fitted, so every difference below is attributable to the model's surroundings, not to the model.

**Three tiers, one task set.** The horizontal comparison places three tiers on the same 114 tasks; each removes one component from the tier above it.

- **Tier 1 — bare model.** One API call per task: no tools, retrieval, knowledge surface, or repair. It receives the task prompt, its declared artifacts, and, for `bug_fix`, the mutated project to repair — task inputs, not domain knowledge — and returns one file bundle, scored as submitted. This tier bounds the model's unaided contribution and is the only one testing premise 1 of §1 in isolation, having no mechanism to compensate for a missing syntax prior.
- **Tier 2 — general harnesses.** Three general-purpose CLI harnesses, reported per system without averaging. They supply tool use, file editing and repair against the compiler's verdict, but no domain grounding: they detect that a project fails to build, and cannot determine what the correct construct is.
- **Tier 3 — AFSIM-Agent.** The full domain harness of §2; its numbers are the ablation reference arm (A0, §4.3) under the same benchmark and scorer.

Tiers 1 and 2 share backbone, prompt, tasks and scorer, so their step isolates the harness. Tiers 2 and 3 share tasks and scorer, so their step isolates what grounding adds beyond general tooling.

All 342 harness cells (3 arms $\times$ 114 tasks) are valid; one (a Codex objective task) produced no submission after six attempts and is counted as a failure in the denominator. The bare-model tier contributes 114 further cells, all answered. Cross-system comparisons are paired on identical tasks. The ablation runs on the 34 executable tasks only; each arm disables one mechanism family, and an independent dose checker confirms zero dose on the consumption side. Small-sample counts are descriptive.

### 4.2 Horizontal comparison across the three tiers

Table 2 reports the main comparison.

Table 2: Horizontal comparison. Struct.$\wedge$run combines the structure and runtime layers; clean run is the runtime layer alone.

| System | Obj. /80 | `bug_fix` /13 | `full_gen` /21 | Exec. /34 | S$\wedge$R | Clean | Behav. |
|---|---|---|---|---|---|---|---|
| Direct LLM (no harness) | 47 (58.8%) | 5 | 1 | 6 (17.6%) | 17.6% | 17.6% | 17.6% |
| Claude Code (Anthropic, 2026) v2.1.282 · 2026-09-24 | 70 (87.5%) | 11 | 4 | 15 (44.1%) | 94.1% | 97.1% | 44.1% |
| Codex CLI (OpenAI, 2026) v0.155.1 · 2026-09-18 | 66 (82.5%) | 11 | 5 | 16 (47.1%) | 94.1% | 97.1% | 47.1% |
| OpenClaw (OpenClaw contributors, 2026) v2026.9.5 · 2026-09-19 | 69 (86.3%) | 10 | 4 | 14 (41.2%) | 91.2% | 94.1% | 41.2% |
| **AFSIM-Agent (ours)** | **71 (88.8%)** | **11** | **9** | **20 (58.8%)** | **100%** | **100%** | **58.8%** |

Each Tier-2 row gives the harness version and release date (UTC); the vendor appears in the citation. Versions are read from each arm's `run_manifest.json`. All five rows use `deepseek-flash` and are not fine-tuned.

Figure 2 plots the two transitions. Results are reported by slice:

- **Objective.** AFSIM-Agent 71/80, Claude Code 70/80, OpenClaw 69/80, Codex CLI 66/80. Per-type breakdown below.
- **Bug fix.** AFSIM-Agent 11/13, Claude Code 11/13, Codex CLI 11/13, OpenClaw 10/13. With only 13 tasks, differences are descriptive.
- **Full generation.** AFSIM-Agent 9/21, Codex CLI 5/21, Claude Code 4/21, OpenClaw 4/21.
- **Direct LLM (no harness).** 47/80 objective, 6/34 executable. Tier 1, with the architecture removed entirely; it bounds the model's unaided contribution.

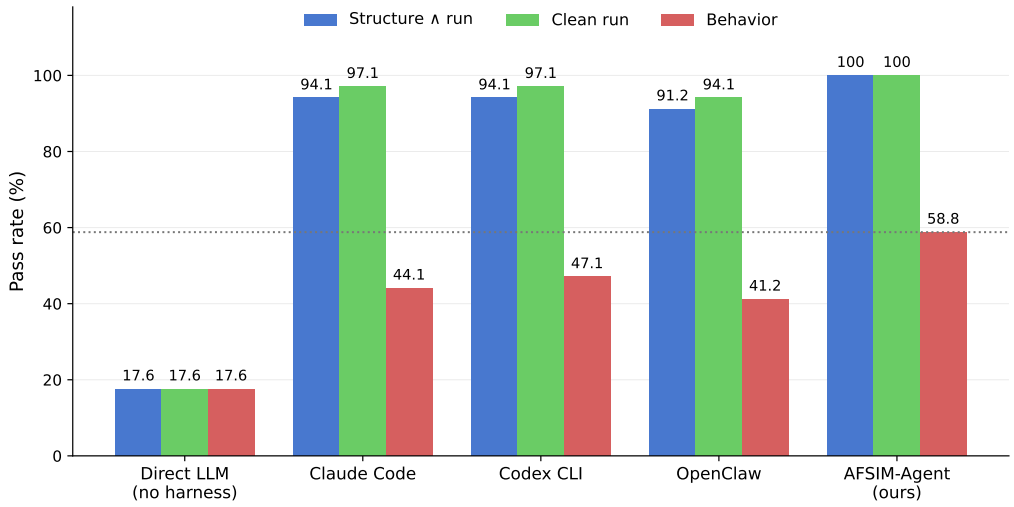


Figure 2: Pass rates at the structure/run and behavior rungs. The direct LLM stops at the first rung; the general harnesses clear it but fall at behavior. **AFSIM-Agent reaches 58.8% at behavior, above all three general harnesses.**

Table 3: Objective track by task type.

| System | qa /20 | choice /20 | fill_blank /20 | code_reading /20 |
|---|---|---|---|---|
| Direct LLM (no harness) | 13 | 15 | 7 | 12 |
| Claude Code (Anthropic, 2026) | 20 | 18 | 18 | 14 |
| Codex CLI (OpenAI, 2026) | 20 | 17 | 16 | 13 |
| OpenClaw (OpenClaw contributors, 2026) | 20 | 17 | 18 | 14 |
| **AFSIM-Agent (ours)** | **20** | **18** | **20** | **13** |

Table 3 gives the per-type breakdown. The level score separates the tiers more sharply. AFSIM-Agent has no cell below L3: every submission runs and produces part of the required behavior. The three general harnesses have 10 cells at L0–L2 (7 fail the structure layer, 3 run cleanly but produce no required behavior) and concentrate at L3 (partial required events).

Table 4: Level-score distribution on the 34 executable tasks.

| System | L0 | L1 | L2 | L3 | L4 | L5 | L6 | Mean level |
|---|---|---|---|---|---|---|---|---|
| Direct LLM (no harness) | 1 | 27 | 0 | 0 | 0 | 0 | 6 | 1.85 |
| Claude Code (Anthropic, 2026) | 2 | 0 | 0 | 12 | 1 | 4 | 15 | 4.41 |
| Codex CLI (OpenAI, 2026) | 2 | 0 | 2 | 10 | 0 | 4 | 16 | 4.41 |
| OpenClaw (OpenClaw contributors, 2026) | 3 | 0 | 1 | 11 | 0 | 5 | 14 | 4.24 |
| **AFSIM-Agent (ours)** | **0** | **0** | **0** | **7** | **0** | **7** | **20** | **5.18** |

The level score runs from L0 (no valid submission) to L6 (strict pass), with intermediate levels marking structure, clean execution, partial behavior, full behavior, and order/time-window satisfaction. AFSIM-Agent has no cell below L3 and reaches L6 on 20 of 34 tasks; the general harnesses concentrate at or below L3. Against the bare model the paired comparison is decisive (15:1, $p = 5.2 \times 10^{-4}$), whereas differences among the three general harnesses are not significant.

AFSIM-Agent leads both tracks: 71/80 on objective tasks and 20/34 executable tasks. The executable lead comes mainly from full generation (9/21 versus 1 and 4–5); bug fixing is nearly tied (11/11/11/10, with 5 for the bare model). Every harnessed system exceeds 91% at structure/runtime but falls below 59% at behavior, so the remaining difficulty is behavioral rather than compilation or execution.




Figure 3: Ablation deltas on `full_generation` (21 tasks) relative to A0 (9/21). **Removing all executable feedback drops A0 to 2/21.**

The direct-LLM row confirms that this is a different failure layer: 26 of its 28 failures are parse/syntax errors and 2 are runtime crashes, with no behavior failures. The harness therefore supplies the main transition from 17.6% to 91–94% at structure/run, while domain grounding supplies the behavior transition from 41–47% to 59%. The two gains are not substitutes, and a zero intent gap is meaningful only when the project actually runs.

### 4.3 Mechanism ablation: executable feedback and specialized mechanisms

Whereas §4.2 compared tiers differing by whole layers, this section varies one mechanism at a time within the top tier, decomposing "domain grounding" into parts. Table 5 reports the deltas. The reference arm A0 scores 11/13 on `bug_fix` and 9/21 on `full_generation`; every batch passed the independent dose check with zero violations.

Tier 1 gives this section a fixed point: it lies below every arm here, so an ablation that measures what it claims should converge on the bare model's position as the architecture is removed.

Table 5: Ablation results by mechanism. Arm labels are paper labels; the mapping to experiment batch labels is released with the data (Reproducibility Statement).

| Arm | Removed mechanism | `bug_fix` (13) | Δ | `full_gen` (21) | Δ | Mean level |
|---|---|---|---|---|---|---|
| **A1** | **All executable feedback** | **6** | **−5** | **2** | **−7** | 2.82 |
| A2 | Full-run feedback (probes only) | 9 | −2 | 5 | −4 | 4.41 |
| A3 | Interaction cell: no LID × probes only | 9 | −2 | 5 | −4 | 4.24 |
| A4 | KB index queries | 11 | 0 | 6 | −3 | 4.88 |
| A5 | Repair-loop discipline | 11 | 0 | 6 | −3 | 4.94 |
| A6 | Honesty audit | 10 | −1 | 6 | −3 | 4.85 |
| A7 | Write-time compile probe | 11 | 0 | 7 | −2 | 5.00 |
| A8 | Skill cards | 10 | −1 | 7 | −2 | 4.71 |
| A9 | Context discipline (LID) | 10 | −1 | 8 | −1 | 5.00 |
| A10 | Static checks + event summary | 11 | 0 | 9 | 0 | 5.21 |
| A11 | Pre-build plan gate | 10 | −1 | 11 | +2 | 5.21 |
| **A0 (ours)** | **— reference (all mechanisms)** | **11** | **0** | **9** | **0** | **5.18** |

Figure 3 shows the deltas.

The paired McNemar test for A1 gives $p = 0.016$; the remaining 1–3-cell differences are point estimates, with failure-composition changes describing how each mechanism acts.

**Executable feedback is the load-bearing mechanism.** With all executable feedback removed, `full_generation` drops from 9/21 to 2/21 and the failure shape changes with it: the reference

arm's failures are almost entirely at the behavior layer — the project runs, only the behavior is missing — whereas A1's include 12 syntax errors and 2 runtime crashes (13 parse/syntax and 4 runtime/crash across both tracks), failing at the more basic task of producing a project that builds. Feedback thus acts along the whole construction chain — syntax, loading, running, behavior — each layer depending on the signal it provides; AFSIM's syntax and semantics lie outside the model's priors, and natural-language reasoning cannot substitute for the compilers and the simulator. A1's mean level falls from the reference 5.18 to 2.82, with 17 cells at L1 (structure passed, runtime failed); its $-7$ cells measure the loss when this gap is fully open.

**Full-run feedback decides whether the agent crosses from "runs" to "satisfies intent."** A2 and A3 keep write-time probes but remove full-run feedback, and both fall to 5/21 on `full_generation`. Most of their projects still compile and run to completion, but the specified behavior does not occur: the event chain does not form, or key events miss their time window. Probes catch local syntax and loading problems but cannot see cross-file duplicate definitions, missing event chains, or contract failures. The failure distribution confirms it: the probe-only arms hold parse errors and crashes to 1–3 cells (against 13 syntax and 4 crashes with no feedback), yet missing_event rises to 16 (reference 12). Probes hold the syntax layer; only the full run supplies behavior-layer signals.

**The knowledge surface and skill cards reduce "runs but behavior absent" failures.** Removing KB index queries turns 3 previously passing `full_generation` tasks into runs whose required behavior does not occur, spanning remote sensing, multi-platform coordination and precise control; removing skill cards turns 2 more, spanning coordination and trajectory prediction. Zeroing the KB and skill doses in the plan window costs each arm one further cell, and both show over-repair failures. These failures share one form: the project is structurally complete and runs cleanly, and the problem is not finding the right construct at generation time — which is what the knowledge surface and skill cards supply.

**Repair, honesty and context mechanisms suppress "stop when it runs."** Removing repair-loop discipline raises missing_event failures from 12 to 16 cells; removing the honesty audit raises them to 17. Both increases sit at the behavior layer with no syntax-layer change: these mechanisms push an already-running project toward intent rather than letting the agent stop at "run finished." Removing the write-time probe raises this class to 14, so early local feedback blocks part of the problems before the behavior layer. The context discipline acts on composition: missing_event falls from 12 to 11 while "other" rises from 2 to 4. Its interaction with feedback depth points the same way: under full feedback, discipline-ON scores 9/21 and OFF 8/21, while under probe-only both score 5/21; the `bug_fix` gains are $+2$ and $+1$. This supports reading the context discipline as preserving the benefits of deep feedback.

**A10 and A11 are positioned by their observed behavior.** A10 (static checks + event summary removed) nets zero — two cells improve, two regress — showing no stable behavior-level benefit on this task set, so it is positioned as an auxiliary level of the feedback ladder. A11 (pre-build plan gate removed) scores 2 cells above reference on `full_generation`, with two countermeasure-type tasks flipping from fail to pass ($p = 0.500$), and one below on `bug_fix`. This cannot show the gate is harmful, but it does show the current implementation does not meet its design intent: it narrows the search space without reliably improving the verdict. The plan gate is not listed as a contribution; §5 discusses its redesign.

## 5 Failure Cases and Limitations

This section reports implementation-level shortcomings observed in experiments. Cases from invalidated batches and audits not yet completed are not listed.

### 5.1 Implementation-level shortcomings

**F1 — The plan gate: a mechanism that does not degrade when switched off.** The pre-build plan gate requires a structured build plan and a deterministic audit before any write tool unlocks. Its ablation is a null result: with the gate removed, `full_generation` scores 11/21 (reference 9/21, $p = 0.500$) and `bug_fix` 10/13 (reference 11/13, $p = 1.00$), so neither benefit nor harm can be claimed. The intervention is narrower than its name, because both arms retain the cognitive-map

injection and prompt-level planning; its protection also overlaps with full-run feedback, and 8 of 21 tasks fail for every arm, leaving little room for significance. We therefore do not claim the gate as a working contribution; future work should make the plan reusable across rounds or merge plan audit with fidelity checking.

**F2 — Implementation boundaries of the verifier and the auditor.** The agent's internal verifier is a prompt-level sub-agent and does not constitute an independent oracle; the fidelity audit is likewise produced by the same agent being evaluated. The agent's own verdicts therefore serve only as evidence about that agent; final verdicts on executable tasks come solely from the frozen evaluator.

**F3 — The remaining failures on full generation are where the implementation falls short.** Of the 21 full-generation tasks in the horizontal comparison, 10 fail for all five systems—the bare model, the three general harnesses, and AFSIM-Agent. The mechanisms raise this slice's pass count from 1 (bare model) and 4–5 (general harnesses) to 9, but over half of the tasks still end at "runs, behavior absent." This is not a sample-size issue: these failures occur on the reference arm with all mechanisms enabled, showing that the current implementation of the feedback ladder, knowledge surface, and repair loop cannot yet cover the constructs and behavior chains these tasks require. Consequently, the 4–6-cell lead on the executable track is of the same order as batch retest noise, and none of the three paired comparisons reaches significance; the path to widening the lead is better implementation (mechanism iteration as discussed under F1), not more tasks.

## 6 Conclusion

Low-resource executable DSLs break the assumptions that make general code agents effective: syntax is not in-distribution, examples are scarce, and ordinary compiler feedback does not decide task intent. The central object is therefore not code generation alone but observation. AFSIM-Agent connects domain priors to executable feedback, preserves decisive verdicts across long work, and repairs against a verified project state; AFSIM-Bench supplies behavior-level verdicts that distinguish running from satisfying intent.

The transferable lesson is that raising the verifier to the behavior level changes what an agent can learn from its environment. This matters for any domain whose correctness is defined over runtime traces, including hardware description languages, physical modeling, PLC logic, and database migrations. A system that only observes compilation or process exit cannot discover whether its output satisfies intent, and additional search or natural-language reasoning does not repair that missing observation. The practical implication is that domain grounding should be evaluated by the verifier it enables, not by its resemblance to a generic agent architecture. Future work should test cross-DSL transfer, synthesize supervision from verification traces, develop learned oracles, make the fidelity audit decidable, and redesign the plan gate around reusable intermediate artifacts.

### AI use statement

Per the ICLR 2027 policy, LLM involvement is disclosed by task type. **Synthetic data generation:** a substantial share of the QA and code-reading objective tasks was generated by LLMs from case families, then reviewed task by task by the authors. **Data cleaning:** knowledge-base cleaning was performed by an LLM cleaning agent with author review. **Implementation:** parts of the harness and evaluation code were written with LLM assistance; all code was reviewed by the authors and is covered by the test suite. **Hypotheses, methodology design, and result interpretation:** done by the authors; LLMs were used only for discussion and literature search. **Figures, writing, and literature search:** LLM-assisted, with all citations verified by the authors. No LLM-generated content is presented as the authors' own original scientific claim without author verification.

### Ethics statement

This paper involves multi-domain modeling and simulation; its potential impact and the handling of it are described below.

**What is released.** The release includes task descriptions, oracle contracts, the event-normalization schema, scoring code, and hashes.

**Task abstraction.** Benchmark tasks test simulation-engineering ability—multi-file project organization, sensor and link configuration, event timing. Tasks contain no real sensitive parameters and no real geographic information.

**Risks and mitigations.** Technology that lowers the cost of building simulation scenarios is dual-sided: it serves engineering research and teaching, and it could also be used for analysis in high-stakes scenarios. The net effect of publication is judged positive, because the paper's core contribution is *measurement*: it shows that systems claiming success on such tasks often fail to achieve the intended behavior—information the domain needs. Nevertheless, the release surface is limited as described, and the full scenario corpus is not published.

**Human subjects.** This research involves no human subjects; benchmark tasks were generated and reviewed entirely by the authors.

**AI use.** See the dedicated statement above.

### Reproducibility statement

The release will include: the frozen task set with tags and manifest, oracle contracts and complete predicate lists, the event-normalization schema, scoring and analysis code, all experiment configuration files including seeds, and a per-table provenance record binding every table to `script + git commit + config hash + command + output hash`. The AFSIM runtime is not redistributed with the benchmark; a mock provider is provided so the scoring pipeline can be exercised end-to-end without the simulator. Also released with the data are the mapping between the anonymized task labels (T01–T34 / Q01–Q80) and the original task identifiers, and the mapping between the paper's ablation-arm labels and the experiment batch labels. Every score is bound to a `benchmark_version / harness_commit / oracle_manifest_hash` triple with a changelog.

## References


Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in ai safety. *arXiv:1606.06565*, 2016.

Anthropic. Claude Code. Agentic coding tool, 2026. URL `https://www.anthropic.com/claude-code`.

Ruisheng Cao et al. Spider2-v: How far are multimodal agents from automating data science and engineering workflows? In *NeurIPS Datasets and Benchmarks*, 2024. doi: 10.52202/079017-3421.

Lee J. Cronbach and Paul E. Meehl. Construct validity in psychological tests. *Psychological Bulletin*, 52(4):281–302, 1955. doi: 10.1037/h0040957.

Yangruibo Ding et al. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *NeurIPS Datasets and Benchmarks*, 2023.

Alexandre Drouin et al. Workarena: How capable are web agents at solving common knowledge work tasks? In *ICML*, 2024.

Xueying Du et al. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. In *ICSE*, 2024.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, and Boris Ginsburg. Ruler: What's the real context size of your long-context language models? In *COLM*, 2024.

Abigail Z. Jacobs and Hanna Wallach. Measurement and fairness. In *ACM Conference on Fairness, Accountability, and Transparency (FAccT)*, 2021. doi: 10.1145/3442188.3445901.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models. In *EMNLP*, 2023.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *ICLR*, 2024.

Liqiang Jing et al. Dsbench: How far are data science agents from becoming data science experts? In *ICLR*, 2025.

Victoria Krakovna, Jonathan Uesato, Vladimir Mikulik, Matthew Rahtz, Tom Everitt, Ramana Kumar, Zac Kenton, Jan Leike, and Shane Legg. Specification gaming: the flip side of ai ingenuity. DeepMind blog, 2020. URL https://deepmind.google/discover/blog/specification-gaming-the-flip-side-of-ai-ingenuity/.

Yuhang Lai et al. Ds-1000: A natural and reliable benchmark for data science code generation. In *ICML*, 2023. URL https://proceedings.mlr.press/v202/lai23b.html.

Fangyu Lei et al. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. In *ICLR*, 2025.

Mingjie Liu et al. Verilogeval: Evaluating large language models for verilog code generation. In *ICCAD*, 2023. doi: 10.1109/ICCAD57390.2023.10323812.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024a. doi: 10.1162/tacl_a_00638.

Tianyang Liu et al. Repobench: Benchmarking repository-level code auto-completion systems. In *ICLR*, 2024b.

Shuai Lu et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *NeurIPS Datasets and Benchmarks*, 2021.

Yao Lu et al. Rtllm: An open-source benchmark for design rtl generation with large language model. In *ASP-DAC*, 2024. doi: 10.1109/ASP-DAC58780.2024.10473904.

Mike A. Merrill et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv:2601.11868*, 2026. doi: 10.48550/arXiv.2601.11868. URL https://arxiv.org/abs/2601.11868.

OpenAI. Codex CLI. Open-source coding agent for the terminal, 2026. URL https://github.com/openai/codex.

OpenClaw contributors. OpenClaw. Open-source agent platform, 2026. URL https://docs.openclaw.ai.

Vaishnavi Pulavarthi et al. Assertionbench: A benchmark to evaluate large-language models for assertion generation. In *NAACL*, 2025. doi: 10.18653/v1/2025.findings-naacl.449.

Zachary S. Siegel et al. Core-bench: Fostering the credibility of published research through a computational reproducibility agent benchmark. *Transactions on Machine Learning Research*, 2024.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. In *Proceedings of the ACM on Software Engineering*, volume 2, pp. 801–824, 2025. doi: 10.1145/3715754.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Diyi Yang. Swe-agent: Agent-computer interfaces enable automated software engineering. In *NeurIPS*, 2024.

Alex L. Zhang. Language model harnesses are compositional generalizers. Personal blog, 2026. URL https://alexzhang13.github.io/blog/2026/harness/.

Yuxuan Zhu et al. Establishing best practices for building rigorous agentic benchmarks. *arXiv:2507.02825*, 2025.

Terry Yue Zhuo et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *ICLR*, 2025.

# A PER-TASK DETAILS

Executable-track ladder codes: `S` = structure layer passed (declared required artifacts complete); `R` = clean run (zero return code and no engine error lines); `B` = all behavior predicates passed; $\star$ = strict ($S \wedge R \wedge B$); `.` = layer not passed; leading `-` = missing declared artifact (strict necessarily fails). Task IDs are anonymized: executable track T01–T34, objective track Q01–Q80; the mapping to original task identifiers is released with the data package (`anonymization_map.csv`).

## A.1 EXECUTABLE TRACK (34 TASKS)

Table 6: Per-task layered verdicts on the executable track.

| Task | Direct LLM | Claude Code | Codex CLI | OpenClaw | **AFSIM-Agent (ours)** |
|---|---|---|---|---|---|
| T01 | SRB* | SRB* | SRB* | SRB* | SRB* |
| T02 | SRB* | SRB* | SRB* | SR. | SRB* |
| T03 | SRB* | SRB* | SRB* | SRB* | SRB* |
| T04 | S.. | SRB* | SRB* | SRB* | SRB* |
| T05 | S.. | SRB* | SRB* | SRB* | SRB* |
| T06 | S.. | SRB* | SRB* | SRB* | SRB* |
| T07 | SRB* | SR. | SR. | SR. | SR. |
| T08 | SRB* | SR. | SRB* | SRB* | SRB* |
| T09 | -.. | SRB* | SRB* | SRB* | SRB* |
| T10 | S.. | -R. | -R. | -R. | SRB* |
| T11 | S.. | SRB* | SRB* | SRB* | SRB* |
| T12 | S.. | SRB* | SRB* | SRB* | SR. |
| T13 | S.. | SRB* | SRB* | SRB* | SRB* |
| T14 | S.. | SRB* | SR. | SRB* | SR. |
| T15 | S.. | SR. | SR. | -.. | SR. |
| T16 | S.. | SRB* | SRB* | SR. | SRB* |
| T17 | S.. | SR. | SR. | SR. | SR. |
| T18 | SRB* | SRB* | SRB* | SRB* | SRB* |
| T19 | S.. | SR. | SR. | SR. | SR. |
| T20 | S.. | SR. | SR. | SR. | SRB* |
| T21 | S.. | SR. | SR. | SR. | SR. |
| T22 | S.. | SR. | SR. | SR. | SRB* |
| T23 | S.. | SRB* | SR. | SR. | SRB* |
| T24 | S.. | SRB* | SRB* | SRB* | SRB* |
| T25 | S.. | SR. | SR. | SR. | SR. |
| T26 | S.. | SR. | SRB* | SR. | SRB* |
| T27 | S.. | SR. | SR. | SRB* | SR. |
| T28 | S.. | -.. | -.. | -.. | SR. |
| T29 | S.. | SR. | SR. | SR. | SRB* |
| T30 | S.. | SR. | SRB* | SR. | SRB* |
| T31 | S.. | SR. | SR. | SR. | SR. |
| T32 | S.. | SR. | SR. | SR. | SR. |
| T33 | S.. | SR. | SR. | SR. | SR. |
| T34 | S.. | SR. | SR. | SR. | SR. |

## A.2 OBJECTIVE TRACK (80 TASKS)

The objective track is a single-layer judgment (regex fullmatch against gold), so each cell is only PASS/fail; "$\varnothing$" means no submission, counted as a failure (§4.1).

Table 7: Per-task results on the objective track.

| Task | Direct LLM | Claude Code | Codex CLI | OpenClaw | **AFSIM-Agent (ours)** |
|---|---|---|---|---|---|
| Q01 | fail | PASS | PASS | PASS | PASS |
| Q02 | PASS | PASS | PASS | PASS | PASS |
| Q03 | fail | PASS | PASS | PASS | PASS |
| Q04 | fail | PASS | PASS | PASS | PASS |
| Q05 | PASS | PASS | PASS | PASS | PASS |
| Q06 | PASS | PASS | PASS | PASS | PASS |
| Q07 | PASS | PASS | PASS | PASS | PASS |
| Q08 | PASS | PASS | PASS | PASS | PASS |
| Q09 | PASS | PASS | PASS | PASS | PASS |
| Q10 | fail | PASS | PASS | PASS | PASS |
| Q11 | PASS | PASS | PASS | PASS | PASS |
| Q12 | fail | PASS | PASS | PASS | PASS |
| Q13 | PASS | PASS | PASS | PASS | PASS |
| Q14 | fail | PASS | PASS | PASS | PASS |
| Q15 | PASS | PASS | PASS | PASS | PASS |

(continued)

| Task | Direct LLM | Claude Code | Codex CLI | OpenClaw | **AFSIM-Agent (ours)** |
|---|---|---|---|---|---|
| Q16 | PASS | PASS | PASS | PASS | PASS |
| Q17 | fail | PASS | PASS | PASS | PASS |
| Q18 | PASS | PASS | PASS | PASS | PASS |
| Q19 | PASS | PASS | PASS | PASS | PASS |
| Q20 | PASS | PASS | PASS | PASS | PASS |
| Q21 | fail | fail | PASS | fail | fail |
| Q22 | PASS | PASS | PASS | PASS | PASS |
| Q23 | PASS | PASS | PASS | PASS | PASS |
| Q24 | PASS | PASS | PASS | PASS | PASS |
| Q25 | fail | fail | fail | fail | fail |
| Q26 | fail | PASS | PASS | PASS | PASS |
| Q27 | PASS | PASS | PASS | PASS | PASS |
| Q28 | PASS | PASS | PASS | PASS | PASS |
| Q29 | fail | PASS | PASS | PASS | PASS |
| Q30 | PASS | PASS | PASS | PASS | PASS |
| Q31 | PASS | PASS | fail | PASS | PASS |
| Q32 | PASS | PASS | fail | fail | PASS |
| Q33 | PASS | PASS | PASS | PASS | PASS |
| Q34 | PASS | PASS | PASS | PASS | PASS |
| Q35 | fail | PASS | PASS | PASS | PASS |
| Q36 | PASS | PASS | PASS | PASS | PASS |
| Q37 | PASS | PASS | PASS | PASS | PASS |
| Q38 | PASS | PASS | PASS | PASS | PASS |
| Q39 | PASS | PASS | PASS | PASS | PASS |
| Q40 | PASS | PASS | PASS | PASS | PASS |
| Q41 | PASS | PASS | PASS | PASS | PASS |
| Q42 | PASS | PASS | PASS | PASS | PASS |
| Q43 | fail | PASS | PASS | PASS | fail |
| Q44 | PASS | fail | PASS | PASS | PASS |
| Q45 | PASS | PASS | PASS | PASS | fail |
| Q46 | PASS | PASS | PASS | PASS | PASS |
| Q47 | fail | PASS | PASS | PASS | fail |
| Q48 | PASS | PASS | PASS | PASS | fail |
| Q49 | PASS | PASS | fail | PASS | PASS |
| Q50 | PASS | PASS | PASS | PASS | PASS |
| Q51 | fail | fail | fail | fail | PASS |
| Q52 | fail | fail | fail | fail | PASS |
| Q53 | fail | fail | fail | fail | fail |
| Q54 | PASS | PASS | fail | fail | PASS |
| Q55 | fail | fail | fail | fail | fail |
| Q56 | PASS | PASS | PASS | PASS | PASS |
| Q57 | PASS | PASS | PASS | PASS | PASS |
| Q58 | fail | PASS | PASS | PASS | PASS |
| Q59 | fail | fail | ∅ | fail | fail |
| Q60 | PASS | PASS | PASS | PASS | PASS |
| Q61 | fail | PASS | PASS | PASS | PASS |
| Q62 | fail | PASS | PASS | PASS | PASS |
| Q63 | fail | PASS | fail | PASS | PASS |
| Q64 | fail | PASS | PASS | PASS | PASS |
| Q65 | fail | PASS | PASS | PASS | PASS |
| Q66 | PASS | PASS | PASS | PASS | PASS |
| Q67 | fail | PASS | PASS | PASS | PASS |
| Q68 | fail | PASS | PASS | PASS | PASS |
| Q69 | fail | fail | fail | fail | PASS |
| Q70 | fail | PASS | PASS | PASS | PASS |
| Q71 | PASS | PASS | PASS | PASS | PASS |
| Q72 | PASS | PASS | PASS | PASS | PASS |
| Q73 | PASS | PASS | PASS | PASS | PASS |
| Q74 | fail | PASS | PASS | PASS | PASS |
| Q75 | fail | PASS | PASS | PASS | PASS |
| Q76 | PASS | PASS | PASS | PASS | PASS |
| Q77 | fail | PASS | fail | PASS | PASS |
| Q78 | PASS | PASS | PASS | PASS | PASS |
| Q79 | PASS | PASS | PASS | PASS | PASS |
| Q80 | fail | fail | fail | fail | PASS |

## B FAILURE-TYPE DISTRIBUTIONS

Per-arm failure-type counts on the 34 executable tasks (label definitions and priority in §G; A10 has one wall-clock-timeout cell with no submission on the `full_generation` track, so $n = 33$). Column aliases in Tables 8 and 9, in the order the columns appear: DL = direct LLM with no harness, CC = Claude Code, CX = Codex CLI, OC = OpenClaw, A0 = reference arm (all mechanisms); A1–A11 are ablation arms under the paper labels. Type codes used in Table 9: $\cdot$ = pass; M = miss-

ing_event; P = parse/syntax; C = runtime/crash; D = include/dependency; O = wrong_event_order; W = wrong_target_state; A = missing_artifact; X = timeout; G = tooling/config; U = budget_exhausted; E = other; ∅ = cell not in that arm's scoring rows (only T31 on A10, timeout with no submission).

### B.1 AGGREGATE DISTRIBUTION

Table 8: Failure-type counts by arm (34 executable tasks). Columns use the type codes defined above, in priority order. Rows list the bare model and the three comparison harnesses first, followed by the reference arm and the ablations in paper-label order.

| Arm | $n$ | P | D | C | M | O | W | A | X | G | U | E | fail | pass |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Direct LLM (no harness) | 34 | 26 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 28 | 6 |
| Claude Code | 34 | 0 | 0 | 0 | 16 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 19 | 15 |
| Codex CLI | 34 | 0 | 0 | 0 | 16 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 18 | 16 |
| OpenClaw | 34 | 0 | 0 | 0 | 17 | 0 | 0 | 2 | 0 | 0 | 0 | 1 | 20 | 14 |
| **A0 (ours, reference)** | **34** | **0** | **0** | **0** | **12** | **0** | **0** | **0** | **0** | **0** | **0** | **2** | **14** | **20** |
| A1 | 34 | 13 | 0 | 4 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 26 | 8 |
| A2 | 34 | 1 | 0 | 2 | 16 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 20 | 14 |
| A3 | 34 | 1 | 0 | 2 | 16 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 20 | 14 |
| A4 | 34 | 0 | 0 | 0 | 15 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 17 | 17 |
| A5 | 34 | 0 | 0 | 0 | 16 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 17 | 17 |
| A6 | 34 | 0 | 0 | 0 | 17 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 18 | 16 |
| A7 | 34 | 0 | 0 | 0 | 14 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 16 | 18 |
| A8 | 34 | 0 | 0 | 0 | 14 | 0 | 1 | 0 | 0 | 0 | 0 | 2 | 17 | 17 |
| A9 | 34 | 0 | 0 | 0 | 11 | 0 | 1 | 0 | 0 | 0 | 0 | 4 | 16 | 18 |
| A10 | 33 | 0 | 0 | 0 | 12 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 13 | 20 |
| A11 | 34 | 0 | 0 | 0 | 11 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 13 | 21 |

### B.2 PER-TASK FAILURE TYPES

Table 9: Per-task failure-type labels.

| Task | DL | CC | CX | OC | **A0** | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | A9 | A10 | A11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T01 | · | · | · | · | · | · | · | · | · | · | M | · | · | · | · | · |
| T02 | · | · | · | M | · | · | · | · | · | · | · | · | · | · | · | M |
| T03 | · | · | · | · | · | · | · | · | · | · | · | · | · | · | · | · |
| T04 | P | · | · | · | · | · | · | M | · | · | · | · | · | · | · | · |
| T05 | C | · | · | · | · | M | · | · | · | · | · | · | · | · | · | · |
| T06 | P | · | · | · | · | · | · | · | · | · | · | · | · | · | · | · |
| T07 | · | E | E | E | E | E | E | · | E | E | E | E | E | E | E | E |
| T08 | · | W | · | · | · | M | · | W | · | · | · | W | W | W | · | W |
| T09 | P | · | · | · | · | M | · | M | · | · | · | · | · | · | · | · |
| T10 | P | M | M | M | · | P | M | · | · | · | · | · | · | · | · | · |
| T11 | P | · | · | · | · | C | · | · | · | · | · | · | · | · | · | · |
| T12 | P | · | · | · | M | · | M | · | M | M | M | · | M | M | M | M |
| T13 | C | · | · | · | · | C | P | P | · | · | · | · | · | · | · | · |
| T14 | P | · | M | · | E | C | · | · | E | · | · | · | E | E | · | · |
| T15 | P | M | M | A | M | P | M | M | M | M | M | M | M | M | M | M |
| T16 | P | · | · | M | · | M | M | · | M | · | M | · | · | · | · | · |
| T17 | P | M | M | M | M | P | M | M | M | M | M | M | M | M | M | M |
| T18 | · | · | · | · | · | · | · | · | · | · | · | · | · | · | · | · |
| T19 | P | M | M | M | M | P | C | M | M | M | M | M | M | · | M | M |
| T20 | P | M | M | M | · | P | M | M | M | M | M | M | M | M | M | · |
| T21 | P | M | M | M | M | M | M | M | M | M | M | · | M | E | M | M |
| T22 | P | M | M | M | · | P | · | M | · | · | · | · | · | · | · | · |
| T23 | P | · | M | M | · | P | · | M | · | · | M | M | · | · | · | · |
| T24 | P | · | · | · | · | · | · | · | · | · | · | · | · | E | · | · |
| T25 | P | M | M | M | M | M | M | M | M | M | M | M | M | M | M | M |
| T26 | P | M | · | M | · | P | C | C | · | M | M | M | M | · | · | · |
| T27 | P | M | M | · | M | P | M | M | M | M | M | M | M | M | M | M |
| T28 | P | A | A | A | M | P | M | M | M | M | · | M | M | M | · | · |
| T29 | P | M | M | M | · | P | M | · | · | M | M | M | · | · | M | · |
| T30 | P | M | · | M | · | P | M | M | M | M | · | · | · | · | · | · |
| T31 | P | M | M | M | M | C | M | M | M | M | M | M | M | M | ∅ | M |
| T32 | P | M | M | M | M | M | M | C | M | M | M | M | M | M | M | M |
| T33 | P | M | M | M | M | M | M | M | M | M | M | M | M | M | M | M |
| T34 | P | M | M | M | M | P | M | M | M | M | M | M | M | M | M | M |

## C RELATED WORK

This appendix expands the positioning paragraph in §1.

**General software-engineering benchmarks.** SWE-bench (Jimenez et al., 2024) and its variants, RepoBench (Liu et al., 2024b), CrossCodeEval (Ding et al., 2023), CodeXGLUE (Lu et al., 2021), DSBench (Jing et al., 2025), CORE-Bench (Siegel et al., 2024), DS-1000 (Lai et al., 2023), BigCodeBench (Zhuo et al., 2025), and ClassEval (Du et al., 2024) evaluate code understanding, repair, or synthesis. Their target languages are in-distribution, their oracles are cheap and largely objective, and their evaluation signals stop at the structure or runtime level; in these settings, passing the test suite is already close to the definition of intent, so the gap between "runs" and "satisfies intent" does not arise. The interval where that gap does arise is the object of study here.

**Agent–computer interfaces and general agents.** SWE-agent (Yang et al., 2024) establishes the agent–computer interface as a first-class design object; Agentless (Xia et al., 2025) shows simple pipelines can be competitive; Terminal-Bench (Merrill et al., 2026) evaluates agents on realistic terminal tasks; WorkArena and BrowserGym (Drouin et al., 2024) offer a symmetric "benchmark + environment" design. These systems are the comparison targets: §4.2 reports their differences from AFSIM-Agent on identical tasks and identical oracles, and they are not treated as baselines to be patched. Because each of them bundles a harness with a model, the bare model is additionally run on the same tasks, so that the harness and the grounding are not credited to one another; the same tiering is what SWE-bench-style leaderboards would need in order to say whether a reported gain came from tooling or from understanding.

**Domain and DSL code generation.** VerilogEval (Liu et al., 2023), RTLLM (Lu et al., 2024), AssertionBench (Pulavarthi et al., 2025), and related HDL work evaluate generated hardware descriptions; Spider 2.0 (Lei et al., 2025) and Spider2-V (Cao et al., 2024) target enterprise data and computer-use workflows. The HDL sub-cluster's evaluation signals stop mostly at compilation or functional simulation, and most of it ships with open-source simulators, making oracles cheap and easy to share. When the runtime is not open-source, a complete "DSL + agent + measurement instrument" delivery remains open; this paper attempts to fill that position.

**Validity methodology.** DS-1000 (Lai et al., 2023) reports its oracle's false-discovery and false-omission rates; the ABC checklist (Zhu et al., 2025) audits benchmarks along result, task, and reporting validity. This methodology is adopted, and pre-release validity is implemented through freeze gates, gold smoke, and oracle grounding; every result is bound to a benchmark version, harness commit, and oracle manifest hash so later revisions remain traceable (§3).

**Scope of claims.** No novelty is claimed in retrieval, ReAct-style loops, self-reflection, prompt engineering, write-time compilation, behavior contracts, or agent–computer interfaces. The claims are: (i) establishing the "runs" vs. "satisfies intent" gap as the central measurement object for low-resource executable DSLs, with corresponding design requirements (§1); (ii) a hierarchical agent organized around these requirements, in which the executable feedback ladder (§F.2) and verification-gated repair (§F.3) are the load-bearing implementations, with ablation showing executable feedback is the clearest load-bearing mechanism (§4.3); and (iii) the instrument required for behavior-level measurement, including the public–hidden contract and freeze-validity gates (§3).

# D AFSIM-AGENT IMPLEMENTATION DETAILS

The knowledge surface contains 3,341 files (≈183 MiB) of grammar references, manuals, and official demo projects, of which about 28.7 MiB is text material. It also contains 140 Markdown skill cards: 93 topic cards, 46 hand-written verb cards, and one README.

Static validation only checks whether a type or command appears in the grammar and documentation and whether the planning reference closure is complete; semantic validity is left to execution. Repair results are stored against the project-state signature and mutation ordinal of the revision that produced them, so a file mutation invalidates the old verdict.

The LID discipline persists raw requirements to disk, converts sub-agent results into handles, normalizes oversized tool results into spill files, folds repeated failures, and maintains a verified knowledge cache. Completion uses normalized event files as behavior evidence, rejects event-shaped logs written by the submission, blocks pre-planted tracks when the oracle requires a sensing chain, and distinguishes `DONE`, `COMPLETED_WITH_CONSTRAINED`, and `NOT_FULLY_VERIFIED`.

# E PROBLEM DEFINITION: EXTENDED ANALYSIS

Submission checks are divided into four kinds. **Structure** checks whether the artifacts are complete and the syntax and include graph are well formed. **Runtime** checks whether the project executes and exits cleanly. **Behavior** checks whether the expected events, their order, counts, and time windows occur in the normalized event trace. **Fidelity** checks whether the artifacts match the specification and whether undeclared deviations are acceptable.

These four kinds answer different questions. Passing structure or runtime only shows that the project can be loaded and executed; it says nothing about intent, a distinction that belongs to construct validity (Cronbach & Meehl, 1955; Jacobs & Wallach, 2021). The frozen benchmark's final executable verdict covers structure, runtime, and behavior; fidelity checks run as agent-side audits and are not counted in the benchmark's final score.

A general coding agent typically sees two kinds of signals directly: the project it wrote and the compile or run results returned by the toolchain. In general-purpose languages these signals usually suffice to locate problems. In a simulation DSL, a clean process exit only means that the run did not trigger a runtime error; it does not mean that the expected behavior occurred. A scenario can produce none of the required event chain and still finish cleanly.

Adding search rounds therefore cannot close this gap. The agent needs more decidable feedback that separates "the project runs" from "the project satisfies intent." At the same time, the feedback must remain visible in the agent's subsequent context: if a critical verdict is lost to compression or truncation, it cannot guide repair even though the environment produced it.

Table 10: Design requirements and corresponding implementations.

| # | Design requirement | Corresponding implementation |
|---|---|---|
| **R1** | **Supply domain priors** | Curated domain knowledge surface |
| **R2** | **Construct executable feedback aligned with intent** | Static checks, write-time compile probes, full simulation |
| **R3** | **Preserve critical verdicts and control signals** | Locally in-distribution (LID) context discipline |
| **R4** | **Close feedback into repair** | Verification-gated repair |
| **R5** | **Constrain the residual bypass space** | Plan audit, numeric fidelity audit, capability audit, post-verification edit guard |
| **R6** | **Produce executable verdicts at the behavior level** | AFSIM-Bench oracle protocol (§3) |

# F AFSIM-AGENT: EXTENDED MECHANISMS

AFSIM-Agent is a hierarchical, self-reflective coding agent that adapts to an unfamiliar DSL inside the harness, without retraining. The control loop has three phases: `base` handles research and routing; `plan` forms a build plan in a read-only environment and passes a deterministic plan audit; `build` generates and repairs the project under write permission, continuously receiving write-time compile probes and full-simulation feedback. The plan-to-build transition is enforced by code rather than prompt convention.

## F.1 THE CURATED DOMAIN KNOWLEDGE SURFACE

R1 requires supplying domain priors that the model lacks. AFSIM-Agent organizes the DSL's grammar, documentation, and runnable examples as a retrievable surface. The model has almost no AFSIM syntax prior, so the retrieval keywords themselves are already approximate guesses at DSL vocabulary; stacking vector matching's semantic approximation on top of that uncertainty risks drifting far from the intended construct. The knowledge surface therefore deliberately avoids both vector indexing and a custom static grammar-analysis layer: it relies on exact and prefix keyword retrieval over a curated corpus.

The surface has three layers. (1) A **read-only document tree**: 3,341 files (≈183 MiB) of grammar references, manuals, and official demo projects, of which about 28.7 MiB is text material. (2) A

**KB index**: keyword-addressable entries distilled from the document tree, retrieved by exact/prefix matching. (3) **Skill cards**: 140 Markdown cards—93 topic cards, 46 hand-written verb cards, and one README—that link engineering tasks to the source files that implement them, loaded by keyword-overlap scoring or explicit invocation.

Skill cards turn a successful retrieval into a reusable engineering recipe, reducing the chance that a failed lookup is misread as a capability boundary. Static validation stays shallow: it only checks whether a type or command appears in the grammar and documentation, and checks the reference closure during planning; semantic validity is left to the runtime. The main limitation is that keyword recall depends on the query, so evaluating the surface must consider whether the right construct is retrievable at concrete failure points.

### F.2 THE EXECUTABLE FEEDBACK LADDER

R2 requires executable feedback aligned with task intent. AFSIM-Agent orders feedback by increasing cost and increasing proximity to intent: earlier signals are cheaper; later signals better separate "the project runs" from "the project satisfies intent."

Table 11: Feedback levels and returned signals.

| Level | Mechanism | Returned signal |
|---|---|---|
| **Static checks** | Check block types, commands, terminators, and the include closure without starting the simulator | Type, command, and reference errors |
| **Write-time compile probe** | After file creation or edit, wrap the target file and its dependencies into a minimal compilable project, and map errors back to source files | Localizable compile errors and structured failure signatures |
| **Full simulation** | Run the complete project; success requires a clean run plus event-log inspection (process exit alone does not count) | Run status, event counts, and behavior verdicts |
| **Audit feedback** | After full simulation, run numeric fidelity, capability-claim, auto-verification, and public-checklist checks | Spec deviations, false capability claims, and unmet public requirements |

### F.3 VERIFICATION-GATED REPAIR

R4 requires feedback to close into repair. Failures are first returned as structured signatures, and repeated failures are folded and counted in context so that repeated output does not flood critical verdicts. The repair loop does not restrict editing itself; what is gated is evidence validity. Every verification result is bound to the project-state signature and mutation ordinal of the revision that produced it; once a file changes, the old verdict is invalidated and must be re-verified before it can support completion.

When the build reaches behavior success on the current revision, a post-verification edit guard blocks purposeless further edits; if the specification still has unmet requirements, the agent must explicitly declare one before editing again. The harness also tracks repeated tool calls, research and repair budgets, and plan-phase stagnation.

### F.4 THE LID CONTEXT DISCIPLINE

R3 requires critical verdicts to remain visible across long-horizon tasks. AFSIM-Agent keeps the main context limited to decomposition strategy, control flow, and normalized state; task requirements, sub-agent reports, and raw tool output are written to files and read back on demand. Each LLM call thereby stays in a similar operating distribution—consistent with the observation that, on isomorphic tasks, every call inside a harness returns in-distribution (Zhang, 2026)—reducing the degradation that follows when context fills with non-control-flow material (Liu et al., 2024a; Hsieh et al., 2024).

The discipline's core invariant is: **compression may drop details, but never verdicts** (Jiang et al., 2023). Before truncating, deduplicating, or aging tool results, the harness first preserves runtime, behavior, fidelity, and contract verdicts, together with deviation lists and repair guidance. The main

mechanisms are: persisting raw requirements to disk, handle-izing sub-agent results, normalizing oversized tool results into spill files, folding repeated failures, and a verified knowledge cache.

### F.5 Fidelity, honesty, and anti-bypass guardrails

R5 requires the agent to constrain the residual bypass space (Amodei et al., 2016; Krakovna et al., 2020). Beyond checking whether the project runs, AFSIM-Agent adds layered guardrails at the planning, generation, and completion phases. **Plan audit** checks whether platforms, payloads, events, the file graph, and DSL commands have explicit grounding. **Numeric fidelity audit** compares key numbers in the specification against the final artifacts item by item; undeclared deviations block completion. **Capability audit** checks whether a claim that "AFSIM does not support a capability" has a counterexample in the knowledge base.

Completion keeps several further anti-bypass checks: behavior evidence comes only from normalized event files; when the oracle constrains a sensing chain, pre-planted tracks cannot substitute for real detection; and path-consistency and required-mechanism checks limit out-of-scope artifacts and deletion-style repair. Completion statuses distinguish `DONE`, `COMPLETED_WITH_CONSTRAINED`, and `NOT_FULLY_VERIFIED`.

## G AFSIM-Bench: Extended Protocol

The 34 executable tasks are bound to 22 case families with runnable reference projects and are scored by executable oracles validated against reference solutions and sabotaged variants. `full_generation` does not provide the reference project to the agent; `bug_fix` inputs are mutated projects. The frozen core contains no `partial_generation` and no tasks scored by an LLM judge.

Each task is given as a JSONL record with a natural-language specification, a task type, and scoring configuration; executable tasks additionally declare required output artifacts, allowed resources, and a hidden oracle reference. Objective tasks are scored by deterministic rules on the task record and the answer alone; executable-task scoring also depends on the frozen evaluator and the AFSIM runtime, but never executes task-supplied scoring code.

Executable tasks are scored in four layers. The **structure** layer checks whether the declared required artifacts exist. The **runtime** layer drives the engine on the task-declared entry file; a clean run requires both a zero return code and no engine error lines. The **behavior** layer evaluates hidden predicates over the normalized event trace; it runs whenever the engine produced a trace, without being gated by the structure layer. The **strict verdict** is the conjunction of structure, runtime, and strict predicates.

Failures fall into a closed label set—parse/syntax, include/dependency, runtime/crash, missing_event, wrong_event_order, wrong_target_state, missing_artifact, timeout, tooling/config, budget_exhausted—with a fixed priority. The benchmark also reports a 0–6 level score per executable task: L0 no valid submission or structure failure; L1 structure passed; L2 clean run; L3 at least one required-behavior predicate passed; L4 all required-behavior predicates passed; L5 all order and time-window predicates passed; L6 strict pass.

Behavior verdicts accept only engine-written event files, plus facts the evaluator derives statically from project text. The evaluator snapshots event files before the run and admits only files new or changed by this run. Event-shaped logs written by the submission itself are refused and recorded. The public part includes the task specification, required artifacts, allowed resources, and event families or predicate categories; the hidden part includes exact thresholds, time windows, event counts, and the reference solution.

The frozen core passes 20 consistency and scorability gates; objective tasks pass a gold smoke check, and all 34 executable-task oracles are grounded by reference runs and validated by reference-solution and sabotage smoke tests. The release boundary includes the task JSONL, oracle contracts and predicate lists, the event-normalization schema, scoring code, and the freeze manifest. Every result is bound to a `benchmark_version / harness_commit / oracle_manifest_hash` triple.

## H Ablation Summary and Boundaries

The ablation yields a layered conclusion. Executable feedback is the load-bearing mechanism, and the remaining mechanisms each contribute behavior-level gains in their own direction on top of it. A1 is the strongest single piece of evidence: removing all executable feedback drops `full_generation` from 9/21 to 2/21, with failure modes regressing wholesale from behavior back to syntax and runtime — confirming the central claim that an agent in a low-resource executable DSL must obtain executable domain feedback. The specialized mechanisms show up as shifts in failure composition rather than as large deltas in pass rate, each in the direction its name implies.

**The deepest cut lands where the bare model already sits.** With the entire feedback ladder removed, the agent's profile resembles Tier 1 rather than a harness: A1 scores 8/34 against the bare model's 6/34, and its failures move to the same layer — 13 parse/syntax and 4 runtime/crash across both tracks, against 26 and 2, with behavior-layer failures absent from both. Two manipulations from opposite directions — removing the ladder from the agent, and the agent from the model — converge on the same rung. That convergence is the strongest available evidence that the feedback ladder, not the rest of the architecture, carries the harness tier; and it is a check the ablation could not perform without a no-harness baseline.

Two boundaries apply. First, with 13 `bug_fix` and 21 `full_generation` tasks, 1–2-cell differences are point estimates, read through failure-composition evidence. Second, the ablation covers the 34 executable tasks only and does not extrapolate to the 80 objective tasks.

## I Threats Bounding the Conclusions

**Construct.** Behavior predicates are derived from reference implementations and may inherit their incidental properties; whether a semantically equivalent but differently implemented correct solution necessarily passes has not been proven.

**External.** Conclusions come from a single DSL; tasks are derived from the official demo suite and inherit its selection bias.

**Model.** All experiments use the same base model (DeepSeek v4.1 Flash, max setting) via API, and absolute scores are bounded by its capability. Models and APIs drift across versions, so every batch archives its model identifier, and results are bound to the benchmark version, harness commit, and oracle manifest hash (§3).