XGENERATOR: SKILL-GUIDED SYNTHESIS OF VERIFIABLE TRAINING DATA FOR A LOW-RESOURCE MODELING LANGUAGE

Anonymous authors
Paper under double-blind review

ABSTRACT

Low-resource domain-specific languages (DSLs) often lack training data linking explicit requirements, executable models, and verification evidence. In industrial system modeling, constructing such data requires compilation, simulation, and requirement checking within a unified workflow. We introduce XGenerator, a skill-guided framework for synthesizing verifiable training data that keeps base-model parameters fixed during generation. Versioned procedural skills organize language specifications, modeling knowledge, and verified experience for on-demand retrieval. The language model produces a compact executable intermediate design describing components, states, and event interactions. Deterministic tools translate it into source code and perform compilation, simulation, and requirement checking against frozen behavioral contracts. Failure evidence guides scoped design repair and revalidation while requirements and shared skills remain fixed. The resulting artifacts link requirements, designs, code, and execution evidence; subsequent skill revisions require separate development validation. We evaluate XGenerator on 80 discrete-event modeling tasks across eight domains and five difficulty levels in X-language, a low-resource DSL. XGenerator improves first-pass requirement satisfaction over prompt-only baselines by 88.75 and 40.00 percentage points on GPT-6-astra and DeepSeek-flash, respectively.With at most one feedback-guided repair, the final requirement satisfaction rate reaches 97.5% and 80.0%. We further construct a model dataset for fine-tuning. Across four Qwen3-8B runs, requirement satisfaction on 410 tasks with executable criteria reaches 68.78–70.24%, compared with 0% for the base model under raw-output evaluation. These results support constructing traceable training data for low-resource modeling DSLs.

1 INTRODUCTION

Large language models offer a new avenue for generating programs from natural language requirements, but for low-resource domain-specific languages (DSLs), the code available for training and evaluation remains limited. In industrial system modeling, what is especially scarce is not isolated syntax examples, but complete models with clear requirements, executable behavior, and evidence-backed results. Such data must simultaneously express system structure, component interactions, and runtime constraints, and its construction and validation rely on domain knowledge and supporting toolchains(Zhang et al., 2021; Abbas et al., 2021; Lange & Atkinson, 2022). How to transform the generative capabilities of general-purpose LLMs into reliable domain training data under this condition is the question this paper addresses(Figure 1).

The generation of industrial modeling data needs to address three levels of issues simultaneously: whether the model source code can pass compilation, whether the model can complete simulation under specified scenarios, and whether the runtime behavior can satisfy requirements. Compilation validation provides the basis for language and implementation validity; simulation validation examines whether dynamic execution can be completed; and requirement validation further checks whether the observed behavior conforms to task constraints. The three form a layer-by-layer connected data construction pipeline. If the generator merely reproduces the numerical values expected by the checker without implementing the corresponding business processes, then even if it passes

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.

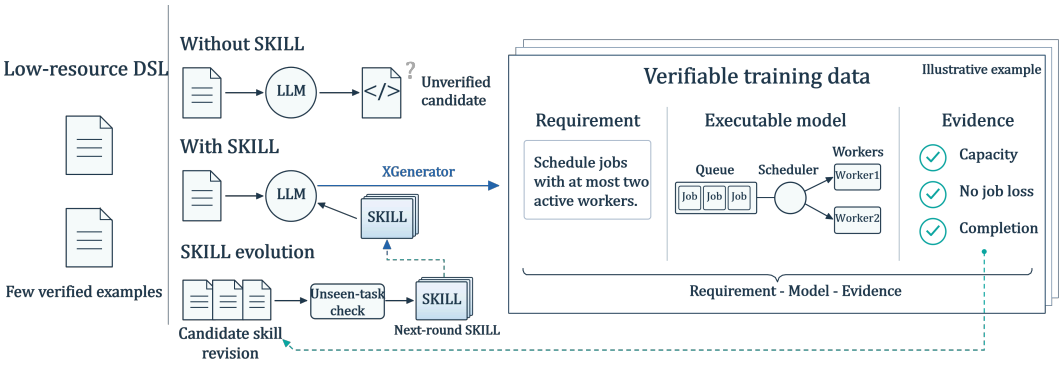

Figure 1: Research overview: skills support verifiable requirement–model–evidence data construction for low-resource DSLs. The schematic contrasts generation without and with SKILL; candidate skill revisions require unseen-task validation before reuse. Base-model parameters remain fixed.

some tests, it may pass erroneous modeling practices down to downstream models. Therefore, validation needs to be embedded into the generation pipeline to provide a basis for candidate screening and failure repair.

Recent work organizes knowledge into skills and improves their use through environmental feedback, with gains depending on tasks, models, and execution frameworks (Zheng et al., 2025; Li et al., 2026). Provenance, layered validation, and experience-based updates offer further building blocks (Shen et al., 2026; Tang et al., 2026). These approaches provide foundations for reusable skills, but how to turn skill-guided generation into verified training data for low-resource modeling DSLs remains insufficiently addressed. In system modeling, successful compilation and simulation do not establish that a model satisfies its original requirements. The challenge is therefore to connect domain-guided design with compilation, simulation, and requirement validation, preserving their correspondence and execution evidence throughout data generation.

To address this gap, we propose XGenerator, which integrates skill-guided modeling with compilation, simulation, and requirement validation to generate traceable training data for low-resource DSLs. A general-purpose coding agent retrieves versioned skills covering language specifications, modeling knowledge, execution procedures, and verified experience. Domain coverage and business-mechanism complexity organize tasks; retrieved skills guide semantic design, while deterministic tools translate and validate that design. Execution feedback supports local repair under frozen requirements, preserving links among requirements, models, and verification evidence. A compact executable intermediate design (CE-ID) connects skill-guided semantic decisions to deterministic code generation, while execution evidence guides local design repair under frozen requirements and validation criteria. Leveraging 80 tasks across eight domains and five difficulty levels, combined with subsystem ablation experiments and synthetic-data fine-tuning, we conducted end-to-end model generation experiments on the low-resource modeling language X-language.

Our contributions are as follows:

- **A verifiable data synthesis framework for low-resource DSLs.** We propose XGenerator, which turns scarce domain knowledge into training candidates through skill-guided design and a unified compilation, simulation, and requirement-validation pipeline, without generation-time parameter updates.
- **Evidence-constrained design and scoped repair for low-resource modeling.** CE-ID separates semantic design from deterministic translation. Execution evidence locates defects and constrains repair under frozen objectives, preserving artifacts for inspection and experience management.
- **Experimental evidence from generation to downstream training.** On 80 X-language tasks, XGenerator achieves 97.5% and 80.0% final requirement satisfaction on Astra and DeepSeek. Ablations examine first-pass reliability; four 8B fine-tunes demonstrate downstream executable delivery and requirement satisfaction.

2 Related Work

Verifiable data synthesis and execution feedback. Instruction bootstrapping expands synthetic supervision (Wang et al., 2022); code-specific approaches further use instruction evolution and open-source snippets (Luo et al., 2023; Wei et al., 2023). Execution evidence connects generated programs to observable behavior through reinforcement learning, test-consistency filtering, and candidate reranking (Le et al., 2022; Chen et al., 2022; Ni et al., 2023). Self-feedback, verbal reflection, and external tests support iterative revision (Madaan et al., 2023; Shinn et al., 2023; Huang et al., 2023); execution and simulated user feedback also support multiround code training data (Zheng et al., 2024). Test coverage limits what passing establishes, and expanded tests reveal previously undetected errors (Liu et al., 2023). We focus on data supply before parameter adaptation: compilation, scenario simulation, and frozen requirement checks form one production workflow for a low-resource modeling language. Its artifacts link requirements, designs, code, and evidence while distinguishing scenario-bounded requirement satisfaction from dataset admission.

Procedural skills and experience-based adaptation. Skills turn task experience into reusable programs, tools, and domain workflows. Skills emerge through environmental exploration (Wang et al., 2023; Zheng et al., 2025), validated task solutions (Yuan et al., 2023), and scientific tool use (Huang et al., 2025). Scientific-resource mining further introduces operational contracts, provenance, and layered validation (Shen et al., 2026). Experience summaries, reasoning memories, and skill documents abstract reusable knowledge from outcomes or interaction traces (Zhao et al., 2023; Ouyang et al., 2025; Yang et al., 2026). Prompt and context optimization instead use metrics or trajectories to adjust pipelines, prompts, and context strategies (Khattab et al., 2023; Agrawal et al., 2025; Zhang et al., 2025). Their adaptation targets differ: MemSkill learns and evolves memory-operation skills, whereas SkillRL jointly updates a skill library and a reinforcement-learning policy (Zhang et al., 2026; Xia et al., 2026). These approaches cannot all be characterized as skill reuse with fixed model parameters. SkillHone and WikiSkill manage revisions through decision histories, persistent knowledge, and validation evidence; SkillsBench finds model-, framework-, and domain-dependent gains (Li & Hu, 2026; Tang et al., 2026; Li et al., 2026). XGenerator fixes base-model parameters during generation, translates domain knowledge into CE-ID, and constrains training-candidate production through compilation, simulation, and frozen checks. Shared-skill evolution is tracked separately from task repair; frozen-version comparisons establish workflow effects, not long-term evolutionary benefits.

3 X Language and Task Representation

X-language integrates system-level modeling and simulation under low-resource conditions. It describes complex products' structure and behavior; its architecture is specified in Chinese national standard GB/T 43270-2023 (State Administration for Market Regulation (SAMR) & Standardization Administration of China (SAC), 2023). Drawing on system and physical modeling, X-language organizes composition and simulation within the Discrete Event System Specification (DEVS) framework. It supports continuous, discret e, and hybrid systems, graphical and textual representations, and compilation and simulation tools (Zhang et al., 2022; Xie et al., 2022; Gu et al., 2024). This toolchain links structure, executable behavior, and results across the path from requirements to evidence.

X-language's deeper syntactic and semantic constraints impose explicit and implicit rules differing from general-purpose programming. Despite existing applications, existing evidence indicates that the pretrained knowledge already available for X-language cannot reliably support large-scale code applications. Two challenges follow. First, whereas SysML and Modelica workflows often emphasize particular modeling or simulation concerns, X-language integrates system-level modeling with multidomain simulation, requiring a longer evaluation chain and thorough feedback. Second, without training, models rely on imitation, interpretation, and transfer from scarce examples. Prompts or skills may support extended reasoning but cannot alone make generation a reproducible functional component. These properties make X-language a demanding testbed for verifiable data synthesis.

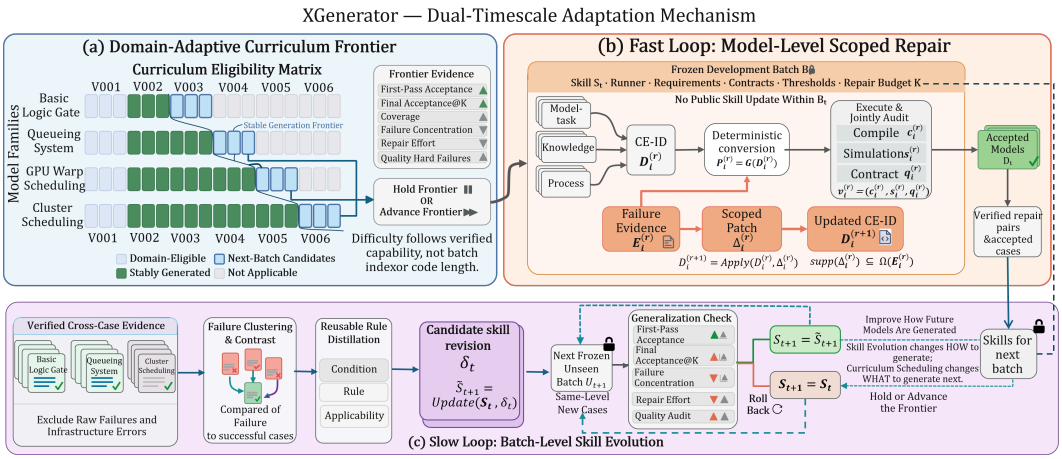

Figure 2: XGenerator: (a) curriculum-guided task selection; (b) skill-guided CE-ID, deterministic translation, execution validation, and evidence-constrained repair; (c) separately validated skill revisions for later data-generation rounds. Shared skills and base-model parameters remain fixed within formal comparisons.

4 METHOD

XGenerator connects task planning, skill-guided CE-ID, deterministic translation, and compilation, simulation, and frozen-contract checks (Figure 2). Failure evidence guides repair and revalidation; verified artifacts support training-data screening and separately validated skill revisions. Appendix A details the evaluation protocol and distinguishes execution validation from training admission.

4.1 TASK FORMULATION AND SKILL-GUIDED DESIGN

A *data-generation round*, indexed by $t = 0, 1, \ldots$, groups tasks processed with a fixed skill version and execution protocol. It is an organizational unit for synthesizing examples, not an optimizer minibatch: no model parameters are updated during generation. Within round t, let $T_i = (R_i, C_i)$, $i \in \{1, \ldots, N_t\}$, denote its N_t tasks. Here R_i is the natural-language requirement and C_i the behavioral contract specifying scenarios, observables, and acceptance criteria. Both are frozen before generation. We omit t from task-level symbols when the round is fixed; S_t consistently denotes that round's skill library, and θ denotes the frozen base-model parameters.

The library organizes language rules, modeling conventions, workflows, and verified experience. We summarize requirement- and stage-based skill selection as $K_{i,p} = \mathrm{Retrieve}(S_t, R_i, p)$. The function takes the library version, requirement, and workflow stage p, and returns the selected skill entries; it does not train a retrieval model. Operationally, the agent first reads CE-ID formats and shared interfaces, then queries indexed rules for the required mechanisms. Applicable verified experience is supplied by the preparation interface. Separate diagnostic lookups supplement these entries by inspecting entities and actions implicated in failures. This progressive disclosure supplies knowledge as decisions require it while keeping S_t unchanged.

CE-ID specifies components, parameters, variables, ports, states, event actions, instances, connections, and simulation entry points. Let r index candidate revisions, with $r = 0$ denoting first-pass generation. The language model F_θ constructs design $D_i^{(0)}$; deterministic translator G converts every revision into source code $P_i^{(r)}$:

$$D_i^{(0)} = F_\theta(R_i, C_i; K_{i,\mathrm{design}}), \qquad P_i^{(r)} = G(D_i^{(r)}). \tag{1}$$

Action bodies contain explicit X-language statements; G supplies no business logic. The design links required functions to implementation entities and checks, connecting inputs, processing, terminal events, and observables updated by actual events rather than scenario-indexed answers. Skills distinguish class-level actions from instance-level event chains, preventing unrelated instances from

forming spurious implementation evidence. Tools check types, ports, connections, actions, and check references before translation. Structural consistency does not establish event reachability, correct guards, or timing, so execution remains necessary. Compared versions jointly alter representations, tools, and workflows, precluding attribution solely to CE-ID.

4.2 Execution Validation and Evidence-Constrained Repair

The contract specifies m_i scenarios, with k_{ij} criteria for scenario j, where $j = 1, \ldots, m_i$ and $\ell = 1, \ldots, k_{ij}$. Successful simulation at revision r yields trace $\tau_{ij}^{(r)}$; criterion $\phi_{ij\ell}$ returns 1 for satisfaction and 0 otherwise. Compilation success is $c_i^{(r)} \in \{0, 1\}$. Indicators apply only to candidates adjudicated at the corresponding layer under the protocol:

$$
\begin{aligned}
s_i^{(r)} &= \begin{cases} 1, & c_i^{(r)} = 1 \text{ and all specified scenarios complete normally,} \\ 0, & \text{otherwise, for adjudicated cases,} \end{cases} \\
q_i^{(r)} &= \begin{cases} 1, & s_i^{(r)} = 1 \text{ and } \phi_{ij\ell}(\tau_{ij}^{(r)}) = 1 \quad \forall j, \ell, \\ 0, & \text{otherwise, for adjudicated cases.} \end{cases}
\end{aligned}
\tag{2}
$$

Sequential checks yield $v_i^{(r)} = (c_i^{(r)}, s_i^{(r)}, q_i^{(r)})$ and diagnostics. Upstream failure fails cumulative indicators; downstream checks remain unexecuted, not independent behavioral defects. Infrastructure errors, missing scores, and incomplete contracts remain unscored outside these definitions. A value $q_i^{(r)} = 1$ denotes passing specified scenarios and criteria. Mechanism plausibility, hard-coding, and out-of-scenario risks still require review before training admission.

R0 permits rule retrieval and static representation checks; all outputs are sealed before unified compilation, simulation, and requirement scoring. Formal comparisons allow at most one feedback repair. R1 uses only the task's sealed implementation and one feedback package, produces one complete candidate, and reruns every scenario. Let $E_i^{(r)}$ denote failure evidence and $\Omega(E_i^{(r)})$ the permitted design-entity edit scope. A scoped patch $\Delta_i^{(r)}$ satisfies

$$
\begin{aligned}
D_i^{(r+1)} &= \mathrm{Apply}(D_i^{(r)}, \Delta_i^{(r)}), \\
\mathrm{supp}(\Delta_i^{(r)}) &\subseteq \Omega(E_i^{(r)}).
\end{aligned}
\tag{3}
$$

Here, supp is the set of modified entities; set containment $\subseteq$ requires every edit to lie within the permitted scope. Hashes bind patches to original designs and actions and verify frozen (R_i, C_i). Scope extensions require justification through existing connections or containment relations. The updated CE-ID is translated again by G and undergoes all prescribed checks. Thus r advances from 0 to 1 within the same generation round t; it does not advance the skill version.

Across generation rounds, the system separately archives revalidated repair pairs and accepted cases with their requirements, diagnostics, changes, and execution evidence. Verified cross-case experience may propose a skill revision δ_t, producing $\widetilde{S}_{t+1} = \mathrm{Update}(S_t, \delta_t)$. It becomes S_{t+1} only if validation on unseen development tasks U_{t+1} meets predefined conditions; otherwise, $S_{t+1} = S_t$. This version-management mechanism excludes infrastructure errors and unverified repair conjectures; frozen-version comparisons do not establish long-term skill-evolution gains.

5 Experiments

5.1 Shared Setup and Metrics

We separately evaluate generation reliability, key subsystems, and training utility. Skill comparisons use 80 tasks: two tasks (A/B) per cell across eight domains and five difficulty levels. Ablations use 40 preselected tasks covering every domain and level (Appendix A). Both follow `unified-codex-v1` and `public-only-v1`: all first-pass outputs (R0) are sealed before scoring; failed tasks receive at most one feedback-based repair (R1), while successful tasks retain R0. Each model-method pair has one batch; stability across repetitions is untested. Comp., Sim., and Req. count tasks that compile, complete all prescribed simulation scenarios, and satisfy frozen requirements,

respectively, corresponding to C/S/Q in the records and $c/s/q$ in Section 4. Req. requires both preceding stages and frozen criteria. Table 1 measures final requirement gains over each model's prompt-only baseline on all 80 tasks. Missing final scores for Luna M3 and Sol M9 count as failures over 80 planned tasks, without altering records. Fine-tuning rescores historical predictions without repair; Table 3 defines missing-output handling.

5.2 Skill Organization and Generation Reliability

All methods receive the task prompt. The reference-material settings are prompt only (M0), prompt plus documentation (M1), prompt plus examples (M2), and prompt plus both (M3); "docs-only" denotes documentation as the sole extra reference source. Skills without tools (M4) uses organized skill rules without method-specific generation tools, while retaining common execution evaluation and feedback. The two *Ours* rows are XGenerator variants: state repair (M8) and action patch (M9). Both combine CE-ID, deterministic translation, and review; the latter adds evidence-bound action patches while retaining state replacement. Table 1 reports all six execution counts and final gains over prompt only for the four models; its descriptive labels correspond to these method identifiers. Luna has no scorable batch for either full variant, and Sol was not evaluated with skills without tools. Each call has a 3,600-second timeout; equal repair opportunities do not imply equal token budgets. Appendix B gives cost and record details.

Domain materials address language barriers, but behavior remains critical. No M0 first-pass output compiles, illustrating prompt-only generation difficulties in this low-resource benchmark. Examples enable Astra M2 to compile and simulate 68 tasks on the first pass, yet only three satisfy requirements. Similarly, DeepSeek M3 simulates 67 tasks but satisfies only nine. Reusing syntax and structure does not ensure resource constraints, event ordering, or recovery behavior. This gap motivates all three verification stages rather than executability-only filtering.

The full workflow improves first-pass reliability and reduces repair demand. M9 improves first-pass requirement satisfaction over M3 by 31.25 and 28.75 percentage points on Astra and DeepSeek, reaching 97.50% and 80.00% finally. Tasks needing repair fall from 34 to nine and from 71 to 48, respectively. On Sol, M8 improves the first-pass rate by 43.75 points over M3 and reaches 75/80 finally, extending the observation to another GPT model. DeepSeek M9 achieves 14 more final requirement successes than M8 despite one fewer simulation success, primarily improving executable models' behavior. Astra M8, however, finishes one task ahead of M9; a more elaborate workflow does not lead at every stage.

Benefits depend on model capability and workflow demands. Luna M3 reaches 23/80 finally, above M2's 16/80 and M4's 3/80. Luna M4 initially compiles 50 tasks but simulates only six; after repair, compilation falls to 18. Additional rules therefore do not ensure reliable dynamic behavior or repair. Luna M8/M9 yield no scorable batches and are neither assigned zero nor used to claim universal gains. Sol M9 initially passes 68 tasks; missing repair scores preclude ranking repair mechanisms from its conservative 68/80 final count. A supplementary Qwen3.8-flash M0 batch ends with one compilation and simulation success and no requirement success; no other methods were run, so it does not measure skill benefits.

Fewer repairs do not imply lower total cost. Astra M9 uses 6.675 million additional R1 tokens versus M3's 10.482 million, but its 74.183 million total tokens are 2.62 times M3's total. Work shifts toward initial design and checking. DeepSeek M9 uses 1.95 times M8's total tokens and 2.00 times its summed call time. More verified candidates require additional generation and checking.

5.3 Subsystem Ablations

M0, M4, and M8 in Table 1 provide an overall comparison as domain organization and specialized design tools are introduced. Astra's first-pass/final requirement counts are 0/0, 45/74, and 67/79; DeepSeek's are 0/0, 24/49, and 27/50. M0 retains public evaluation, and M4 retains skills and feedback. M4→M8 adds several capabilities; its gain cannot be attributed to one tool.

On the same 40 tasks, we further compare full M9 (AE7) with AE8, which removes CE-ID, translation, and design-dependent diagnostics and patches; AE9, which removes enhanced static diagnostics and mandatory behavioral self-review; AE10, which uses ordinary design revision; and AE11,

Table 1: Generation reliability on 80 tasks per group. Initial and final columns report compilation, simulation (Sim.), and requirement (Req.) successes out of 80. $\Delta_{\mathrm{PO}} = 100(n^{\mathrm{Req}}_{\mathrm{final}} - n^{\mathrm{Req}}_{\mathrm{PO,final}})/80$ is the final Req. gain over the same model's prompt-only baseline, in percentage points (pp); green shading indicates its magnitude. Final results allow at most one repair.

Method	**Initial**			**Final**			Δ_{PO}
	Compile	Sim.	Req.	Compile	Sim.	Req.	(pp)
GPT-6-astra (reasoning: low)							
Prompt only	0/80	0/80	0/80	0/80	0/80	0/80	0.00
Prompt + docs	0/80	0/80	0/80	2/80	2/80	2/80	↑ 2.50
Prompt + examples	68/80	68/80	3/80	77/80	77/80	44/80	↑ 55.00
Prompt + docs + examples	51/80	51/80	46/80	76/80	76/80	74/80	↑ 92.50
Skills without tools	50/80	50/80	45/80	77/80	77/80	74/80	↑ 92.50
Ours: state repair†	72/80	72/80	67/80	80/80	80/80	79/80	↑ 98.75
Ours: action patch	75/80	75/80	71/80	80/80	80/80	78/80	↑ 97.50
DeepSeek-flash (reasoning: medium)							
Prompt only	0/80	0/80	0/80	0/80	0/80	0/80	0.00
Prompt + docs	1/80	1/80	0/80	29/80	28/80	7/80	↑ 8.75
Prompt + examples	31/80	31/80	2/80	56/80	55/80	9/80	↑ 11.25
Prompt + docs + examples	68/80	67/80	9/80	80/80	80/80	34/80	↑ 42.50
Skills without tools	62/80	62/80	24/80	76/80	74/80	49/80	↑ 61.25
Ours: state repair	74/80	74/80	27/80	79/80	79/80	50/80	↑ 62.50
Ours: action patch	71/80	70/80	32/80	79/80	78/80	64/80	↑ 80.00
GPT-5.6-luna (reasoning: low‡)							
Prompt only	0/80	0/80	0/80	0/80	0/80	0/80	0.00
Prompt + docs	0/80	0/80	0/80	2/80	1/80	1/80	↑ 1.25
Prompt + examples	18/80	6/80	4/80	32/80	23/80	16/80	↑ 20.00
Prompt + docs + examples*	26/80	19/80	8/80	41/80	39/80	23/80	↑ 28.75
Skills without tools	50/80	6/80	1/80	18/80	7/80	3/80	↑ 3.75
GPT-5.6-sol (reasoning: medium)							
Prompt only	0/80	0/80	0/80	0/80	0/80	0/80	0.00
Prompt + docs	0/80	0/80	0/80	24/80	24/80	17/80	↑ 21.25
Prompt + examples	70/80	70/80	12/80	79/80	79/80	44/80	↑ 55.00
Prompt + docs + examples	66/80	66/80	30/80	78/80	78/80	64/80	↑ 80.00
Ours: state repair†	78/80	78/80	65/80	80/80	80/80	75/80	↑ 93.75
Ours: action patch*	78/80	77/80	68/80	68/80	68/80	68/80	↑ 85.00

*Missing final scores count as failures: 3 for Luna with docs and examples, 12 for Sol with action patches; these conservative final counts do not imply observed repair failures. †Recovery/package exceptions: Appendix B. ‡Luna reasoning effort is verified as low except for the docs-and-examples archive, where it is unavailable.

which removes the explicit frozen-experience workflow. AE6 is an M8 version reference. All use GPT-6-astra and at most one repair; Appendix C specifies retained capabilities and versions.

The design-translation subsystem reduces first-pass delivery failures. AE7 and AE8 achieve first-pass requirement-satisfaction rates of 95.0% and 70.0%; their compilation and simulation counts are 39 and 29. Eleven of AE8's 12 failures fail compilation. The difference primarily arises at language implementation, supporting the division between model-led semantic design and tool-led translation, rather than isolating CE-ID's effect. AE9 compiles and simulates all 40 tasks but fails requirement checks on four, showing behavioral review's separate purpose. Its net gap from AE7 is only two tasks and requires more runs to assess.

Saturated final rates obscure first-pass reliability and repair burden. All groups reach 40/40 after feedback, but AE8 repairs 12 tasks versus AE7's two; the full workflow primarily reduces rework. Its 33.722 million total tokens nevertheless exceed AE8's 17.899 million. AE10 and AE7 start from different failures; equal final rates do not establish repair equivalence. AE11 also cannot isolate the benefit of retrieved experience: AE7's first-pass automatic queries return no matches, and

Table 2: Ablation successes out of 40. Δ_{init} is the initial Req. difference from full M9 (AE7), in pp. AE11's task scores are complete but its summary is provisional. Further details: Appendix C.

Configuration	**Initial**			**Final**			Δ_{init}
	Compile	Sim.	Req.	Compile	Sim.	Req.	(pp)
AE7: full M9	39/40	39/40	38/40	40/40	40/40	40/40	0.00
AE8: no design subsystem	29/40	29/40	28/40	40/40	40/40	40/40	↓ 25.00
AE9: no delivery review	40/40	40/40	36/40	40/40	40/40	40/40	↓ 5.00
AE11: no frozen experience	38/40	38/40	36/40	40/40	40/40	40/40	↓ 5.00
AE6: full M8	40/40	40/40	40/40	40/40	40/40	40/40	↑ 5.00
AE10: ordinary design repair	36/40	36/40	33/40	40/40	40/40	40/40	↓ 12.50

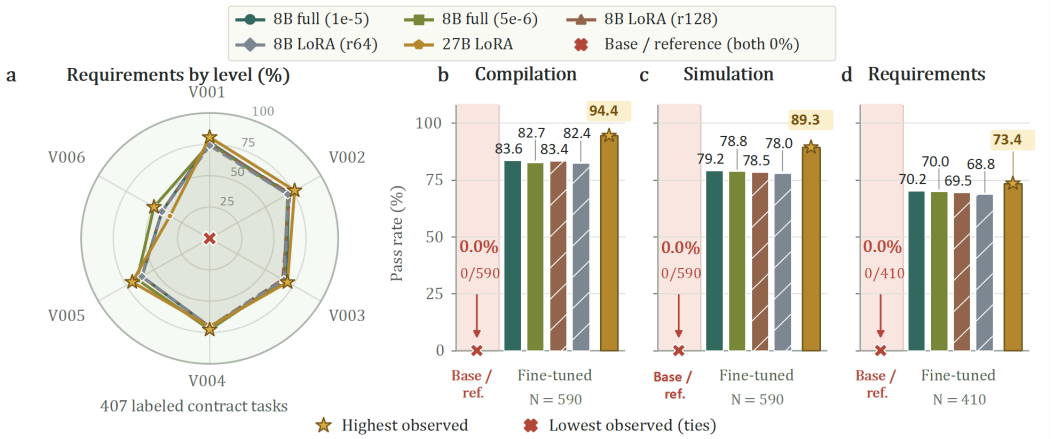

Figure 3: Fine-tuning evaluation of Qwen3-8B and Qwen3.8-27B. (a) Six curriculum levels in the fine-tuning corpus, distinct from the five-level generation benchmark: $N = 119, 114, 93, 44, 23, 14$ (407 labeled contract tasks). (b–d) Overall rates: compilation/simulation $N = 590$; requirements $N = 410$, including three unlabelled contract tasks. Both archived base/reference groups score zero under raw-output evaluation. Three missing 27B predictions remain in planned denominators. The 27B reference identity is unverified; cross-scale differences are descriptive.

only one of six repair-stage matches is recorded as adopted. These ablations primarily inform design, translation, and feedback; frozen-version comparisons do not establish long-term skill-evolution gains.

5.4 Training Utility of Synthetic Data

To assess downstream training utility, we assembled 5,893 candidate models, including 4,097 discrete-event models with contract records covering 5,083 scenarios and 36,646 checks. Excluding one probe without requirements yielded 4,712 training, 590 validation, and 590 test samples. We performed two full-parameter and two LoRA fine-tuning runs on Qwen3-8B, plus one LoRA run on Qwen3.8-27B. Appendix D details data provenance, training settings, repair records, and verification boundaries. Compilation and simulation are evaluated over all 590 test tasks; requirement satisfaction is evaluated only on the 410 discrete-event tasks with complete executable criteria.

Of 590 test tasks, 410 discrete-event tasks have 525 public scenarios and 3,709 checks; 180 lack complete requirement criteria. Evaluation passes raw predictions byte-for-byte to the compiler, including fences and explanatory text, measuring directly usable delivery. Each 8B fine-tune provides 590 outputs; 27B LoRA provides 587. Table 3 distinguishes verified, pending, and missing results.

Training improves both executable delivery and requirement satisfaction. Across the four 8B fine-tunes, compilation rates are 82.37%-83.56%, simulation rates 77.97%-79.15%, and requirement satisfaction 68.78%-70.24% on the 410 eligible tasks. Both full fine-tuning and LoRA produce sub-

Table 3: Evaluation as delivered. Δ_{base}: Req. gain over 8B base (pp); the unverified 27B reference precludes a matched gain. Pending: simulation succeeds but requirement criteria are absent. Missing outputs retain planned denominators; they are not observed failures.

Output group	**Execution evaluation**			**Coverage**		Δ_{base}
	Compile	Sim.	Req.	Pending	Missing	(pp)
8B base	0/590	0/590	0/410	0/180	0/590	0.00
8B full FT, 10^{-5}	493/590	467/590	288/410	108/180	0/590	↑ 70.24
8B full FT, 5×10^{-6}	488/590	465/590	287/410	100/180	0/590	↑ 70.00
8B LoRA, $r = 128$	492/590	463/590	285/410	97/180	0/590	↑ 69.51
8B LoRA, $r = 64$	486/590	460/590	282/410	95/180	0/590	↑ 68.78
27B reference	0/590	0/590	0/410	0/180	0/590	0.00
27B LoRA	557/590	527/590	301/410	146/180	3/590	↑ 73.41

stantial numbers of samples passing execution and frozen criteria, supporting use beyond immediate generation-time repair. Requirement counts differ by at most six tasks; single runs cannot establish a stable ranking of training methods. 27B LoRA passes requirement checks on 301 tasks (73.41% of the planned subset), a supplementary observation at another scale. Its reference identity is not fully established, precluding a clean causal estimate of scaling or fine-tuning alone. Figure 3 also reveals uneven difficulty: V006 requirement satisfaction spans 35.7%–50.0% across the five settings, on only 14 tasks.

Delivery format explains part of the gain but does not replace content verification. Fences occur in 527/590 8B base outputs and 590/590 27B reference outputs, but not in the fine-tuned outputs; logs record backtick parsing failures. The other 63 unfenced 8B outputs also fail compilation, so format is not the sole explanation. Results support improved end-to-end delivery without isolating format and modeling-content gains. Item-level split isolation and loaded test weights still require independent verification; these findings do not guarantee absence of leakage or semantic correctness in arbitrary scenarios.

6 Discussion and Conclusion

XGenerator connects procedural skills, compact executable intermediate designs, and layered validation in a data-production pipeline for low-resource DSLs. Its central contribution is to make compilation, scenario simulation, and frozen requirement checks part of generation and repair, while preserving requirements, designs, code, and evidence as a reviewable training unit. Skills make dispersed domain knowledge callable; CE-ID separates semantic decisions from deterministic implementation.

Skill comparisons support improved generation reliability across the evaluated models, ablations identify the contribution of the design-translation subsystem, and fine-tuning results connect synthetic-data production to executable downstream outputs. Together, these findings support constructing useful domain supervision without updating base-model parameters during generation. Their scope remains bounded: passing public contracts does not establish correctness in all scenarios; experiments cover one language and single batches; and training isolation and long-term skill evolution need further validation.

Future work has three priorities. First, generation and validation remain costly; more selective retrieval, capability-aware decomposition, and learning from verified designs and repairs should be investigated to support less capable models. Second, requirement-driven intermediate design and validation warrant finer treatment of ambiguity, component interactions, timing, exceptional conditions, and scenario coverage. Strengthening these connections is central to both data quality and failure diagnosis. Third, evaluation with additional models, including GLM and Kimi, should characterize applicability and the trade-off between capability, cost, and pass rates. These directions extend the present contribution: a traceable route from scarce domain knowledge to verifiable candidates and downstream training use.

AI USE STATEMENT

We used generative AI within XGenerator to generate synthetic modeling data, as described in the method and experiments. We also used generative AI tools to assist with manuscript polishing, literature checking. The authors are responsible for verifying all AI-assisted content, including references, analyses, and scientific claims, and take full responsibility for the final manuscript and reported results.

REPRODUCIBILITY STATEMENT

The task definitions, scoring semantics, method variants, cost accounting, training settings, and representative execution records are detailed in Appendices A–E. Reported statistics were recomputed from archived task-level records. Missing evaluations and recovery exceptions are disclosed in Appendix B; the limits of training-data isolation, checkpoint provenance, and validation coverage are specified in Appendix D. These details define the scope of the reported results without assuming that unverified records are complete.

REFERENCES

Messaoud Abbas, Renaud Rioboo, Choukri-Bey Ben-Yelles, and Colin F. Snook. Formal modeling and verification of UML Activity Diagrams (UAD) with FoCaLiZe. *Journal of Systems Architecture*, 114:101911, 2021. doi: 10.1016/j.sysarc.2020.101911. URL https://doi.org/10.1016/j.sysarc.2020.101911.

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. arXiv preprint arXiv:2507.19457, 2025. URL https://arxiv.org/abs/2507.19457.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code Generation with Generated Tests. arXiv preprint arXiv:2207.10397, 2022. URL https://arxiv.org/abs/2207.10397.

Pengfei Gu, Zhen Chen, Lin Zhang, Yuteng Zhang, Kunyu Xie, Chun Zhao, Fei Ye, and Yiran Tao. X-SEM: A modeling and simulation-based system engineering methodology. *Journal of Manufacturing Systems*, 74:198–221, 2024. doi: 10.1016/j.jmsy.2024.01.013. URL https://doi.org/10.1016/j.jmsy.2024.01.013.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation. arXiv preprint arXiv:2312.13010, 2023. URL https://arxiv.org/abs/2312.13010.

Xu Huang, Junwu Chen, Yuxing Fei, Zhuohan Li, Philippe Schwaller, and Gerbrand Ceder. CASCADE: Cumulative Agentic Skill Creation through Autonomous Development and Evolution. arXiv preprint arXiv:2512.23880, 2025. URL https://arxiv.org/abs/2512.23880.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. arXiv preprint arXiv:2310.03714, 2023. URL https://arxiv.org/abs/2310.03714.

Arne Lange and Colin Atkinson. Multi-level modeling with LML: A contribution to the multi-level process challenge. *Enterprise Modelling and Information Systems Architectures*, 17(6):1–36, 2022. doi: 10.18417/emisa.17.6. URL https://emisa-journal.org/emisa/article/view/280.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. arXiv preprint arXiv:2207.01780, 2022. URL https://arxiv.org/abs/2207.01780.

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, Chujun Tao, Binxu Li, Xuandong Zhao, Hejia Geng, Xiaojun Wu, Junwei Zhou, Xiaokun Chen, Hanwen Xing, Yubo Li, Qunhong Zeng, Di Wang, Yuanli Wang, Roey Ben Chaim, Penghao Jiang, Haotian Shen, Luyang Kong, Xinyi Liu, Runhui Wang, Xuanqing Liu, Jiachen Li, Xin Lan, Yueqian Lin, Wengao Ye, Junwei He, Songlin Li, Yue Zhang, Yipeng Gao, Yijiang Li, Ze Ma, Liqiang Jing, Tianyu Wang, Kaixin Li, Yiqi Xue, Haoran Lyu, Yizhuo He, Yuchen Tian, Shutong Wu, Bowei Wang, Yixuan Gao, Bo Chen, Litong Liu, Sikai Cheng, Jiajun Bao, Shuaicheng Tong, Shuwen Xu, Terry Yue Zhuo, Tinghan Ye, Qi Qi, Miao Li, Longtai Liao, Zelin Tan, Chang Shi, Xilin Tang, Srinath Tankasala, Boqin Yuan, Yaoyao Qian, Jianhong Tu, Chenguang Wang, Yizhou Sun, Wei Wang, Aaron Taylor, Ziyue Yang, Changkun Guan, Zhikang Dong, Xinyu Zhang, Steven Dillmann, Han-chung Lee, and Dawn Song. SkillsBench: Benchmarking How Well Agent Skills Work Across Diverse Tasks. arXiv preprint arXiv:2602.12670v4, 2026. URL https://arxiv.org/abs/2602.12670v4.

Zhiwei Li and Yong Hu. SkillHone: A Harness for Continual Agent Skill Evolution Through Persistent Decision History. arXiv preprint arXiv:2606.08671, 2026. URL https://arxiv.org/abs/2606.08671.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. arXiv preprint arXiv:2305.01210, 2023. URL https://arxiv.org/abs/2305.01210.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. arXiv preprint arXiv:2306.08568, 2023. URL https://arxiv.org/abs/2306.08568.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative Refinement with Self-Feedback. arXiv preprint arXiv:2303.17651, 2023. URL https://arxiv.org/abs/2303.17651.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-Tau Yih, Sida Wang, and Xi Victoria Lin. LEVER: Learning to Verify Language-to-Code Generation with Execution. In *Proceedings of ICML*, volume 202 of *PMLR*, pp. 26106–26128, 2023. URL https://proceedings.mlr.press/v202/ni23b.html.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling Agent Self-Evolving with Reasoning Memory. arXiv preprint arXiv:2509.25140, 2025. URL https://arxiv.org/abs/2509.25140.

Shuaike Shen, Wenduo Cheng, Mingqian Ma, Alistair Turcan, Martin Jinye Zhang, and Jian Ma. SKILLFOUNDRY: Building Self-Evolving Agent Skill Libraries from Heterogeneous Scientific Resources. arXiv preprint arXiv:2604.03964, 2026. URL https://arxiv.org/abs/2604.03964.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. arXiv preprint arXiv:2303.11366, 2023. URL https://arxiv.org/abs/2303.11366.

State Administration for Market Regulation (SAMR) and Standardization Administration of China (SAC). GB/T 43270-2023: Integrated modeling language for collaborative design of complex products - X language architecture, 2023. URL https://openstd.samr.gov.cn/bzgk/std/newGbInfo?hcno=A9BE5A540F6E6E9C719CD4DE540EEE2F. Published November 27, 2023; effective June 1, 2024.

Liyan Tang, Cyrus Rashtchian, Chun-Sung Ferng, Andrew Tomkins, Da-Cheng Juan, and Tu Vu. WikiSkill: Compiling Agent Experience into Persistent Knowledge for Skill Evolution. arXiv preprint arXiv:2608.27454, 2026. URL https://arxiv.org/abs/2608.27454.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An Open-Ended Embodied Agent with Large Language Models. arXiv preprint arXiv:2305.16291, 2023. URL https://arxiv.org/abs/2305.16291.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-Instruct: Aligning Language Models with Self-Generated Instructions. arXiv preprint arXiv:2212.10560, 2022. URL https://arxiv.org/abs/2212.10560.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering Code Generation with OSS-Instruct. arXiv preprint arXiv:2312.02120, 2023. URL https://arxiv.org/abs/2312.02120.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. SkillRL: Evolving Agents via Recursive Skill-Augmented Reinforcement Learning. arXiv preprint arXiv:2602.08234, 2026. URL https://arxiv.org/abs/2602.08234.

Kunyu Xie, Lin Zhang, Yuanjun Laili, and Xiaohan Wang. XDEVS: A hybrid system modeling framework. *International Journal of Modeling, Simulation, and Scientific Computing*, 13(2): 2243001, 2022. doi: 10.1142/S1793962322430012. URL https://doi.org/10.1142/S1793962322430012.

Yutao Yang, Junsong Li, Qianjun Pan, Bihao Zhan, Yuxuan Cai, Lin Du, Jie Zhou, Kai Chen, Qin Chen, Xin Li, Bo Zhang, and Liang He. AutoSkill: Experience-Driven Lifelong Learning via Skill Self-Evolution. arXiv preprint arXiv:2603.01145, 2026. URL https://arxiv.org/abs/2603.01145.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi R. Fung, Hao Peng, and Heng Ji. CRAFT: Customizing LLMs by Creating and Retrieving from Specialized Toolsets. arXiv preprint arXiv:2309.17428, 2023. URL https://arxiv.org/abs/2309.17428.

Haozhen Zhang, Quanyu Long, Jianzhu Bao, Tao Feng, Weizhi Zhang, Haodong Yue, and Wenya Wang. MemSkill: Learning and Evolving Memory Skills for Self-Evolving Agents. arXiv preprint arXiv:2602.02474, 2026. URL https://arxiv.org/abs/2602.02474.

Lin Zhang, Fei Ye, Kunyu Xie, Pengfei Gu, Xiaohan Wang, Yuanjun Laili, Chun Zhao, Xuesong Zhang, Minjie Chen, Tingyu Lin, and Zhen Chen. An Integrated Intelligent Modeling and Simulation Language for Model-based Systems Engineering. *Journal of Industrial Information Integration*, 28:100347, 2022. doi: 10.1016/j.jii.2022.100347. URL https://doi.org/10.1016/j.jii.2022.100347.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. arXiv preprint arXiv:2510.04618, 2025. URL https://arxiv.org/abs/2510.04618.

Yizhe Zhang, Gregor Hoepfner, Joerg Berroth, Gerwin Pasch, and Georg Jacobs. Towards holistic system models including domain-specific simulation models based on SysML. *Systems*, 9(4):76, 2021. doi: 10.3390/systems9040076. URL https://doi.org/10.3390/systems9040076.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM Agents Are Experiential Learners. arXiv preprint arXiv:2308.10144, 2023. URL https://arxiv.org/abs/2308.10144.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. SkillWeaver: Web Agents can Self-Improve by Discovering and Honing Skills. arXiv preprint arXiv:2504.07079, 2025. URL https://arxiv.org/abs/2504.07079.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. OpenCodeInterpreter: Integrating Code Generation with Execution and Refinement. arXiv preprint arXiv:2402.14658, 2024. URL https://arxiv.org/abs/2402.14658.

A TASKS, PROTOCOL, AND METHOD VERSIONS

The appendices supplement the main results with evaluation definitions, method differences, costs, training evidence, and failure cases. Comp. (compilation), Sim. (simulation), and Req. (requirement satisfaction) correspond to C/S/Q in the original records. Req. counts candidates that pass the prescribed scenarios and criteria; it does not denote overall data-quality admission. Slash-separated entries report counts, not ratios, unless a denominator is explicitly shown. DS abbreviates DeepSeek-flash in the cost table.

A.1 GENERATION, VALIDATION, AND ADMISSION

The main framework (Figure 2) separates semantic design from deterministic translation and execution. During a data-generation round, requirements, behavioral contracts, and shared skills remain fixed. A repaired CE-ID is translated into a new candidate and reevaluated on all prescribed scenarios; previous execution results do not certify the revised source. Requirements, designs, source versions, and execution records are retained together so that a passing result can be traced to the candidate actually evaluated.

Execution success and training admission are distinct. Compilation, simulation, and requirement scores establish only the corresponding checks. They do not certify mechanism fidelity, absence of hard-coded responses, or behavior outside the specified scenarios. The corpus-level evidence in Appendix D distinguishes structured contract verification from less complete historical validation. Formal method comparisons keep shared skills fixed and do not measure longitudinal skill evolution.

A.2 TASK COVERAGE AND SCORING SEMANTICS

The skill benchmark contains 10 tasks per domain, with variants A and B at each level L1-L5. The ablation benchmark contains 5 tasks per domain, selecting A in odd-numbered domains and B in even-numbered domains. Tasks are independent and do not receive solutions to preceding tasks.

Table 4: Domain coverage. L1-L5 denote intended application difficulty, not an interval scale.

Domain	Description	Skill / ablation tasks
D01	Aviation operations and onboard systems	10 / 5
D02	Automotive powertrains and intelligent vehicles	10 / 5
D03	AI computing and large-model serving	10 / 5
D04	Communication networks and edge connectivity	10 / 5
D05	Advanced manufacturing and semiconductor production	10 / 5
D06	Emergency medicine and medication safety	10 / 5
D07	Logistics, ports, and cold-chain supply chains	10 / 5
D08	Power grids and resilient infrastructure	10 / 5

The benchmark revision is 20260912-v3.1. Every task includes nominal, boundary_stress, and compound_exception scenarios with fixed identifiers, seeds, workloads, and exceptions. Parameters and checks are specified by each task's frozen Contract. Observables use the last valid value or a prescribed reducer; event observables prefixed with time_ use first_one_time to obtain the global time of their first value of 1. Simulation termination does not imply application-level completion, and missing series cannot be replaced with zeros. There are no hidden scenarios. Mechanism fidelity and hard-coding checks are separate quality criteria; unknown outcomes are not treated as passes.

A.3 VERIFIED METHOD INVENTORY

Table 5 maps the descriptive labels in the main results to their method identifiers. Capabilities refer to the package used in each recorded batch. Method identity follows the package frozen for the actual batch; later maintenance is not retroactively credited as an evaluated capability. M0 retains the common evaluator and one feedback opportunity. M4 excludes method-specific generation tools while retaining skills and common execution validation.

Table 5: Compared methods and available evidence. Combinations without a scorable batch are not assigned zero scores.

Method	Verified design	Result coverage
M0	Prompt only; direct source generation	Astra, DeepSeek, Luna, Sol
M1	Frozen documentation; direct source generation	Astra, DeepSeek, Luna, Sol
M2	Frozen examples; direct source generation	Astra, DeepSeek, Luna, Sol
M3	Documentation and examples; direct source generation	All four models; missing Luna scores counted as failures
M4	Skill-v0; stage routing and organized rules	Astra, DeepSeek, Luna
M8	CE-ID, action/behavior review, state patches, unified delivery	Astra, DeepSeek, Sol
M9	Instance-level event chains, candidate diagnosis, hash-anchored action patches	Astra, DeepSeek, Sol; missing Sol scores counted as failures

B Skill-Comparison Costs and Evaluation Exceptions

Table 1 reports initial and final compilation, simulation, and requirement counts for all 25 scored model–method groups: 2,000 evaluations of the same 80 frozen requirements. These are repeated task instances across methods, not 2,000 independent tasks. Each group has one batch. Calls have a 3,600-second timeout and a concurrency limit of 4; overall rates are not pooled across models. Table 6 reports costs. The number scheduled for repair is 80 minus the initial requirement count. All scheduled repairs have evaluator scores except Luna M3 (69 of 72) and Sol M9 (0 of 12), whose final-score treatment is specified below.

B.1 Paired Differences and Limits of the Results

For DeepSeek's final M8/M9 requirement outcomes, 44 tasks pass both methods, 20 pass only M9, 6 pass only M8, and 10 pass neither. The aggregate improvement therefore does not imply monotonic improvement on every task. Each difficulty stratum contains only 16 tasks; Figure 4 describes heterogeneity rather than establishing stable rankings.

B.2 Missing Records, Recovery, and Cost Accounting

Luna M3 has final scores for 77 tasks, with known Comp./Sim./Req. counts of 41/39/23. The 3 missing tasks count as failures in the denominator of 80. Sol M9 initially scores 78/77/68; the remaining 12 tasks enter repair, but their R1 evaluator scores are absent from the current archive. Retaining R0 for the 68 tasks requiring no repair and counting the rest as failures yields final counts of 68/68/68. Static review and source delivery do not substitute for execution scores, and the missing-score penalty does not imply 12 observed repair failures. R1 replaces R0 for every task in the repair set; the better round is not selected separately for each metric. Luna M8/M9 remain n/a because neither has a scorable batch.

Astra M8 includes exceptional initial retries on 3 tasks and method-package migrations during repair on 5 tasks; it therefore does not establish performance under a strictly frozen method version. Luna M0/M1 have complete task-level evaluator scores, but their auxiliary summaries remain provisional. Sol M8 retains recovery events and final scores for all 15 repaired tasks. Infrastructure incidents are not directly counted as model failures. These recovery and version exceptions limit comparisons with strictly frozen runs.

Usage is unknown for one DeepSeek M3 call. Astra M8 uses 53,434,112 tokens for valid deliveries. Its table entry instead uses the invocation ledger: 45,197,228 initial tokens plus 11,326,392 repair tokens, including two repair calls without a delivery, for 56,523,620 tokens in total. Both scopes exclude three exceptional R0 attempts. Sol M8 reports 98,235,025 tokens, with unknown usage for one additional capacity-error call. Luna M3 costs are partial; Sol M9 has no complete cumulative ledger. Incomplete costs cannot be ranked against complete totals. Durations sum individual calls and are not elapsed parallel runtime. Displayed columns are rounded independently.

Table 6: Skill-comparison costs. Tokens are in millions; time is summed call duration in hours. Cached input is included in input tokens. Provisional, partial, and subtotal entries preserve the original ledger scope; unknown usage is not imputed as zero.

Model / method	Initial	R1 increment	Total tokens	Hours
Astra/M0	5.971	6.967	12.938	5.487
DS/M0	10.281	11.298	21.578	4.210
Astra/M1	19.248	25.631	44.879	7.506
DS/M1	52.796	50.761	103.557	12.303
Astra/M2	9.575	18.272	27.847	7.820
DS/M2	28.473	50.904	79.377	13.986
Astra/M3	17.831	10.482	28.313	6.124
DS/M3	67.025	76.744	Subtotal 143.769	16.878
Astra/M4	24.224	8.158	32.382	6.109
DS/M4	55.452	44.813	100.266	11.332
Astra/M8	45.197	11.326	Exception subtotal 56.524	8.551
DS/M8	150.338	144.441	294.779	15.041
Astra/M9	67.509	6.675	74.183	8.833
DS/M9	394.682	178.707	573.388	30.069
Luna/M0	9.981	11.067	Provisional 21.048	4.610
Luna/M1	25.059	19.129	Provisional 44.188	7.422
Luna/M2	13.784	16.628	30.412	5.717
Luna/M3	14.596	18.169	Partial 32.765	5.923
Luna/M4	21.601	22.442	44.043	7.429
Sol/M0	7.176	8.702	15.878	8.872
Sol/M1	32.094	42.792	74.886	15.757
Sol/M2	13.787	29.970	43.756	10.923
Sol/M3	32.559	41.849	74.408	12.750
Sol/M8	74.948	23.287	Subtotal 98.235	17.116
Sol/M9	-	-	Not aggregated	-

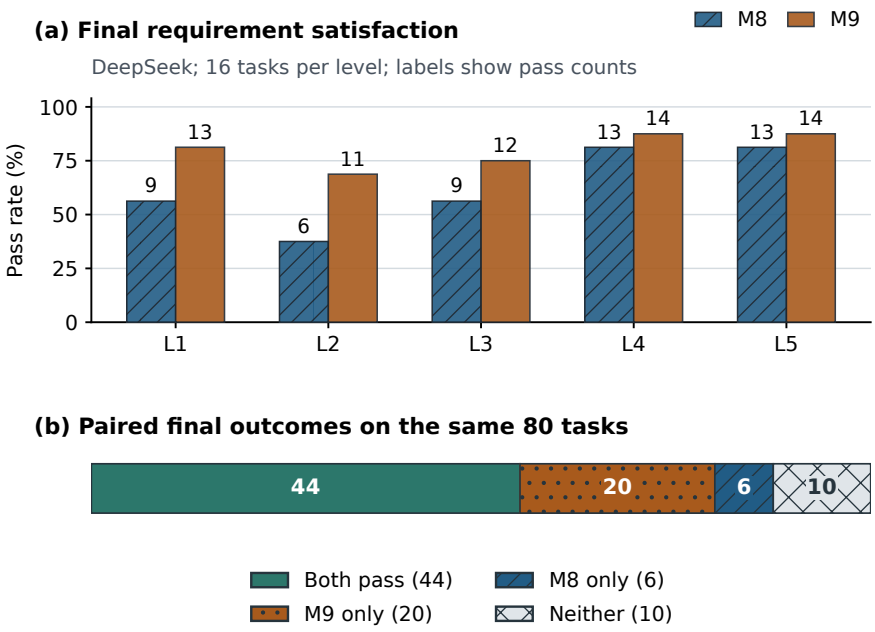

Figure 4: Task-level heterogeneity for DeepSeek. Top: final requirement satisfaction by difficulty, with 16 tasks per level. Bottom: the final pass sets for the same 80 tasks. Both color and texture distinguish conditions. No error bars across repeated runs are shown.

C Ablated Capabilities, Paired Outcomes, and Costs

Table 2 contains the execution counts. The tables below specify what each ablation removes, the scored repair workload and cost, and paired first-pass requirement outcomes. AE11 has complete task-level scores but a provisional auxiliary summary. AE6 is a whole-version reference, and AE10 uses independently generated initial candidates.

Table 7: Removed and retained capabilities. When an ablation removes dependent capabilities together, results are interpreted at the subsystem level.

Group	Changed capabilities	Retained capabilities and interpretation
AE7	None; full M9	Common design reference
AE8	Removes CE-ID, translation, and design-dependent diagnosis and patches; writes source directly	Retains language rules, frozen experience, general self-checks, interfaces, and one feedback opportunity; not an isolated representation ablation
AE9	Removes enhanced static diagnosis, event-chain and completion-path review, and mandatory behavioral self-checks	Retains minimal structural/lexical checks, CE-ID, translation, and local repair; not a removal of all checks
AE11	Removes explicit retrieval, injection, and adoption requirements for frozen experience	Retains basic rules and the remaining workflow; neither knowledge erasure nor an ablation of automatic evolution
AE6	Uses full M8	Whole-version reference
AE10	Replaces evidence localization, candidate-cause comparison, and constrained patches with ordinary design revision	Retains CE-ID, translation, review, experience, and one feedback opportunity; initial candidates are generated independently

Table 8: Ablation repair workload and costs. Repairs are scored/scheduled tasks. Tokens are in millions; hours sum call durations. AE11's auxiliary ledger is provisional, while its task scores are complete. Values are rounded independently.

Group	Repairs	Initial tokens	R1 tokens	Total tokens	Hours
AE7	2/2	32.305	1.417	33.722	4.423
AE8	12/12	13.729	4.170	17.899	4.397
AE9	4/4	17.605	2.171	19.775	3.204
AE11	4/4	29.388	2.908	32.296	4.354
AE6	0/0	21.189	0.000	21.189	3.731
AE10	7/7	29.601	4.136	33.737	4.587

Table 9: Paired first-pass requirement satisfaction on the same 40 tasks. An AE7 win means only AE7 passes; a loss means only the comparison passes. Ties include matching successes or failures. Δ_{init} is comparison minus AE7, in percentage points. Each configuration has one batch.

Comparison	AE7 wins	AE7 losses	Ties	Δ_{init} (pp)
AE8	12	2	26	-25.0
AE9	4	2	34	-5.0
AE11	4	2	34	-5.0
AE6	0	2	38	$+5.0$
AE10	7	2	31	-12.5

AE7's initial automatic-experience queries yield zero hits and zero adoptions on all 40 tasks. Its two R1 tasks return 6 experience entries in total, with only 1 recorded adoption. Thus, the initial AE11 difference cannot be attributed to retrieved experience. Final scores of 40/40 for all groups do not establish equivalent repair mechanisms because their initial failure sets differ. Domain and difficulty strata contain only 5 and 8 tasks, respectively, and do not establish stable rankings.

D Data Construction and Training Evidence

D.1 Data Scale, Validation Coverage, and Splits

The archived candidate manifest contains 5,893 models: 4,097 discrete-event, 1,788 hybrid, and 8 continuous models. Source SHA-256 hashes match the production compilation manifests, and all candidates have successful compilation, successful simulation, and nonempty outputs. The discrete-event candidates have structured contract records covering 5,083 scenarios and 36,646 checks. These statistics describe retained candidates after repair, not first-pass success rates; directory-level totals containing library components are not counted as independent models.

Excluding one discrete-event probe without a requirement section leaves 5,892 samples: 4,096 discrete-event, 1,788 hybrid, and 8 continuous models. The archived preprocessing manifest records a type-stratified, group-preserving split with seed 42 into 4,712 training, 590 validation, and 590 test samples. The group key and actual split contents have not been independently audited for item overlap or near duplicates; byte-level uniqueness alone does not establish semantic diversity or train–test isolation.

The skill-comparison benchmark evaluates discrete-event modeling; the broader training corpus also contains historical hybrid and continuous models. Hybrid records include retrospectively added requirements and conditional checks, while continuous models lack comparable contracts. These records do not establish application of the same frozen-contract production procedure across all model types. Compilation and simulation can be assessed for the full test set, but requirement satisfaction is reported only for the 410 discrete-event tasks with complete executable criteria.

D.2 Training Settings and Checkpoints

Tables 10 and 11 report the shared 8B settings and run-specific records. Training telemetry documents optimization; it does not by itself verify which weights produced the archived test outputs.

Table 10: Training and generation settings for the 8B comparisons in the main text. For 27B training telemetry, only exported curves are available; generation predictions are evaluated separately. Its configuration is not inferred from the run display name.

Item	Settings and evidence scope
Training methods	Two full fine-tuning runs and two LoRA runs; run-specific settings in Table 11
Optimization and batches	3 epochs; per-device batch=1; gradient accumulation=2; AdamW torch fused; weight decay=0.01; cosine schedule; warmup=0.03; max grad norm=1; seed=42
Sequences and objective	Cutoff length=32,768; packing=false; train on prompt=false; archived maximum length=25,429; overflow=0
LoRA modules	Dropout=0.05; q/k/v/o projections and gate/up/down projections
Checkpoints	Evaluation and saving every 100 steps; all minimum validation losses at step 500; completion records at step 885; recovery status in Table 11
Hardware and numerics	8 NVIDIA H100 80GB GPUs; bf16/tf32; unrecorded compute or token counts are not inferred from hardware
8B generation interface	qwen3_nothink; thinking disabled; complete .xl source only; historical predictions for all 590 tasks evaluated unchanged
8B generation parameters	Greedy decoding; seed=42; repetition penalty=1; maximum input=24,576 tokens; maximum new tokens=16,384; eval batch=1
27B and reference group	Existing predictions and author-exported plots retained; complete training settings, raw history, and loaded-weight identity remain incomplete

Table 11: The four 8B training runs and their evaluation output groups. Minimum validation losses come from actual telemetry and are not execution pass rates. Each group contains one run.

Output group	Run ID	Learning rate	r/alpha	Min. val. loss	Completion and recovery
8B full 1e-5	`q3full1e5r1`	1e-5	-	0.040772	Finished; restored step 500
8B full 5e-6	`q3full5e6r1`	5e-6	-	0.041964	Finished; restored step 500
8B LoRA r128	`q3l128a256lr1e4r2`	1e-4	128/256	0.042035	Finished; best weights restored from filesystem
8B LoRA r64	`q3l64a128lr2e4r2`	2e-4	64/128	0.041379	Online status: crashed; archived recovery; telemetry ends at step 865

The full fine-tuning runs take 2,427.8 and 2,418.0 seconds, respectively; LoRA-r128 takes 1,760.4 seconds. No comparable native duration summary is available for r64, and no value is imputed. Training settings, output directories, test manifests, and prediction hashes link the stages, but the hash of the weights loaded for testing was not independently retained. These links do not establish completed item-level isolation checks or authenticated checkpoint identity.

D.3 Training and Optimization Process

The first four panels of Figure 5 use the actual histories of the same four runs in Table 11, preserving every observed point. Hollow diamonds for the two full fine-tuning runs indicate evaluations after restoring the best checkpoint. The final r128 evaluation precedes restoration; the best value restored for r64 is not plotted as an additional evaluation. Online training and optimization telemetry for r64

ends at step 865, and periodic validation ends at step 800; completion records are not used to extend the curves.

The 27B panels reproduce training-loss and learning-rate curves exported from the run named `qwen3.8-27b-lora-8192`. Raw step-level history and the export smoothing settings are unavailable, so no numerical history is inferred from the images. Validation loss and gradient norm are not available; the separately archived `eval/steps_per_second` curve measures throughput and cannot substitute for validation loss.

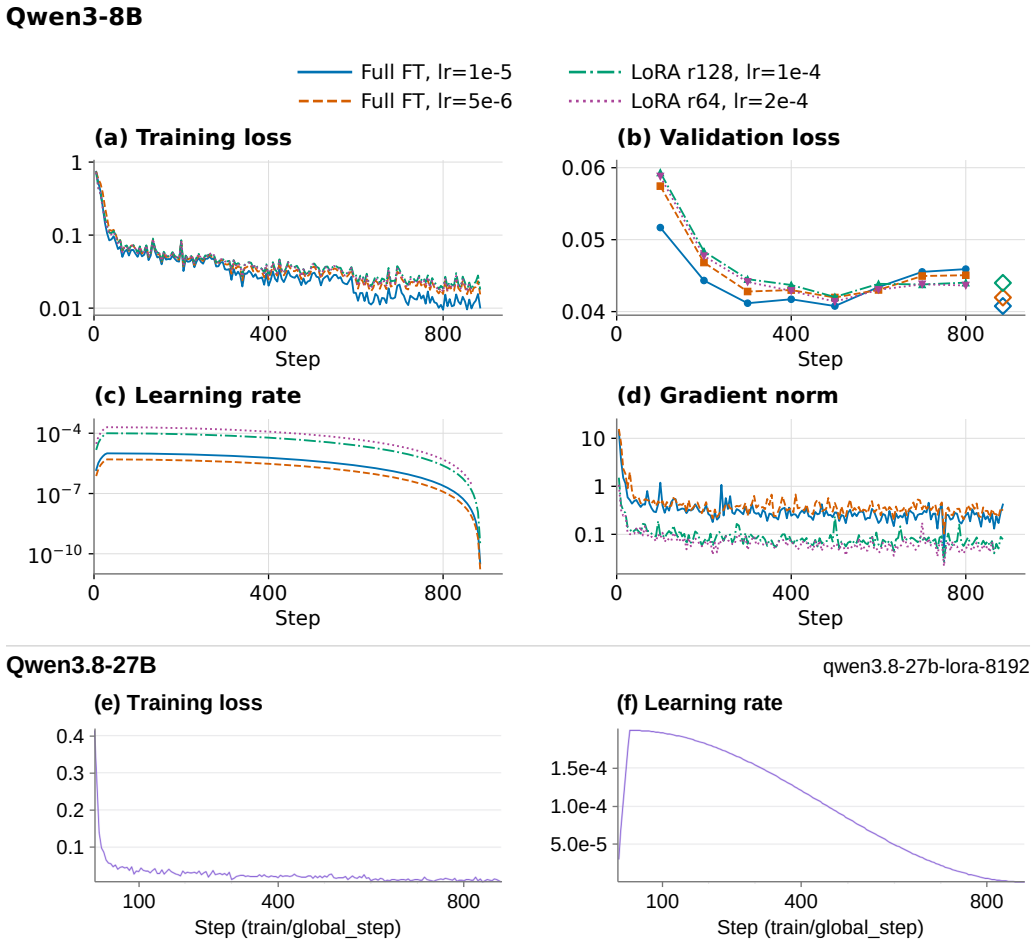

Figure 5: Training evidence for 8B and 27B. The first two rows show 8B training loss, validation loss, learning rate, and gradient norm. The last row shows exported 27B training-loss and learning-rate curves side by side. The 8B training-loss, learning-rate, and gradient-norm panels use logarithmic vertical axes; the others use linear vertical axes. All horizontal axes show training steps. These curves document optimization; generation execution results are reported in Figure 3 and Table 3.

D.4 Validation Coverage by Model Type

The 590 test tasks comprise 410 discrete-event, 179 hybrid, and one continuous task. The 410 discrete-event tasks contain 525 public scenarios and 3,709 checks. Of these tasks, 407 have curriculum labels (V001–V006); three are unlabelled and remain in the overall requirement denominator. These six corpus levels differ from the five benchmark difficulty levels (L1–L5). Missing predictions retain the planned denominators: the three missing 27B predictions comprise one task of each model type.

Table 12: Complete counts by test type. Discrete columns report Comp./Sim./Req.; hybrid columns report Comp./Sim. Hybrid tasks lack complete requirement criteria, so unassessed outcomes are not labeled requirement failures. The single continuous task fails compilation in every group with an available prediction; its prediction is missing for 27B LoRA. Missing counts are discrete / hybrid / continuous.

Output group	Discrete (410 tasks)	Hybrid (179 tasks)	Missing
8B base	0/0/0	0/0	0/0/0
27B reference (unverified)	0/0/0	0/0	0/0/0
8B full 1e-5	385/359/288	108/108	0/0/0
8B full 5e-6	388/365/287	100/100	0/0/0
8B LoRA r128	391/366/285	101/97	0/0/0
8B LoRA r64	391/365/282	95/95	0/0/0
27B LoRA	404/381/301	153/146	1/1/1

E Real Evidence Chains and Failure Cases

The following examples come from the formal DeepSeek M9 batch and illustrate a successful local repair and a repair that remains unsuccessful. Each pairs a task ID with an actual patch and recorded execution outcome. They explain the evidence chain rather than estimate failure prevalence.

E.1 Passing After a Local Timing Repair

In flexible-assembly task D05-L3-B, scenario X02 requires completing six orders from initial inventories X=4 and Y=6 after replenishing X=2, without occupying the robot during material shortages. R0 compiles and simulates, but completedOrders is 0 rather than the expected 6. Diagnosis compares replenishment timing, inventory/rework, and observation as candidate causes. The patch changes only Scenario/SupplyDelay's hold from 1 to 3, preserving the send action. The frozen X02 requirement specifies the shortage-and-replenishment sequence but no exact replenishment timestamp; the patch changes the scenario implementation without changing the requirement or executable checks. After deterministic translation, the R1 source includes:

```
statehold(3);
...
send(supplyX02, 2);
transition(Done);
```

The recorded R1 evaluator scores pass compilation, simulation, and requirements. The scenario output contains the following actual record; X02 starts at global time 500:

```
504.12999999999988,oX02_completedOrders,6
```

This chain supports passing the frozen criteria after a local change. However, the feedback lacks direct event traces, so early replenishment remains a root-cause hypothesis rather than an established explanation. The result does not establish semantic correctness in all scenarios.

E.2 A Timing Criterion Still Fails After Patching

Dynamic-batching task D03-L2-B requires the first batch to start at local time 0 and finish at 2. Frozen criterion N01_04 uses first_one_time to check the start flag against the expected time 0; R0 instead yields 2. The patch anchors to ObservationBench.Observing/receive:0 and the original action hash, inserting a self-transition after setting the flag:

```
oN01_time_batch1Start = 1;
transition(Observing);
```

R1 still compiles and simulates, but the same criterion reports expected=0 and actual=2.0. The actual trace is 0 at time 0 and becomes 1 only at time 2. The patch therefore does not resolve the timing discrepancy. Existing records cannot distinguish an incorrect application-level start time from a mismatch in observation or sampling semantics; marking the patch addressed does not imply requirement satisfaction.

Fine-tuning failures occur at different stages. In the 8B full fine-tuning run with learning rate 10^{-5}, AttributeTransformation case 0017 (V001) lacks the scalar observable `sink.inputSum`, while case 0021 (V005) lacks the scenario-injection parameter `sys.transform.invalidId`. The 8B base output for case 0002 begins with a Markdown code fence and triggers a lexical error. These examples distinguish requirement-observation, scenario-execution, and delivery-format failures, motivating stage-specific evidence rather than an aggregate score alone.