PROGRAMGEN: SCALABLE ENVIRONMENT CONSTRUCTION FOR LONG-HORIZON AGENT TRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

Building an entire software repository from scratch is a challenging long-horizon task for software agents, requiring sustained exploration, implementation, execution, and debugging across many files. Training agents for such zero-to-one repository construction is essentially a data problem: realistic repositories are abundant, but scalable training tasks with reliable executable verification and high-quality long-horizon trajectories are not. We introduce PROGRAMGEN, a repository-to-environment compiler for constructing such training data from existing software repositories. PROGRAMGEN supports two complementary forms of reconstruction: PROGRAMGEN-DOC retains the original tests as a hidden verifier and synthesizes an aligned specification, while PROGRAMGEN-EXEC retains the original program as an execute-only behavioral oracle and synthesizes behavioral tests from its observable behavior. Through agentic construction, PROGRAMGEN produces 10,000 environments across five programming languages and 4,000 verifier-selected trajectories of up to 512K tokens. Training Qwen3.5-35B-A3B-Base on only 1,000 PROGRAMGEN-DOC trajectories raises NL2Repo pass rate from 0.5% to 23.0%. The supervision also transfers beyond repository reconstruction and remains effective after post-training, improving Qwen3.6-35B-A3B from 28.8% to 31.8% on NL2Repo and from 69.6% to 75.4% on SWE-bench Verified.

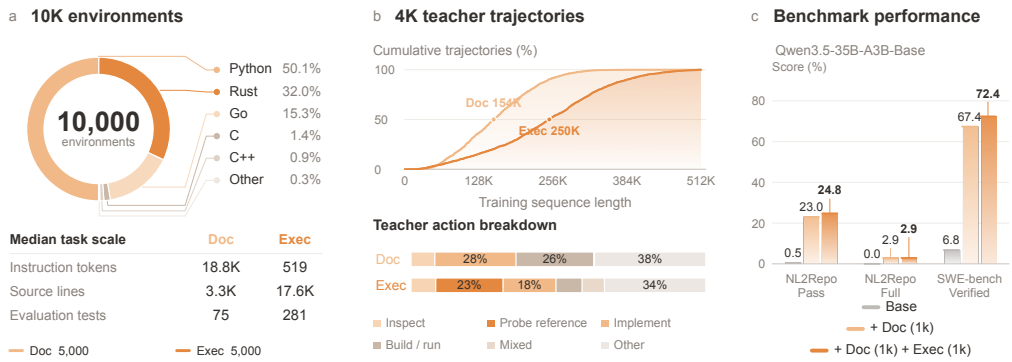

Figure 1: ProgramGen environment diversity, teacher trajectories, and downstream performance.

1 INTRODUCTION

Software agents are increasingly expected not only to modify existing codebases, but to build complete software systems from scratch. Whole-repository construction is a particularly demanding long-horizon setting: an agent must sustain planning, exploration, implementation, execution, and debugging across many files before receiving an end-to-end signal of whether the resulting system is correct (Ding et al., 2025; Yang et al., 2026b). Despite rapid progress in coding agents, this zero-to-one setting remains far from solved (Ding et al., 2025; Yang et al., 2026b).

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.

A central bottleneck is training data. Public repositories provide abundant realistic software, but turning them into reconstruction environments requires hiding the implementation, providing sufficient task information, and preserving automatically executable verification. Existing work follows two main regimes: retaining the original tests and synthesizing a specification (Zhao et al., 2025; Ding et al., 2025; Zhao et al., 2026), or retaining the original program as a behavioral reference and synthesizing tests (Yang et al., 2026b; Chen et al., 2026; Kim et al., 2026). In either case, source isolation is essential: agents may otherwise recover the hidden implementation through package registries, version history, or unintended reference access, turning reconstruction into answer recovery rather than software development (Kahn et al., 2025; Yang et al., 2026b; Kim et al., 2026; Aleithan et al., 2024; Zhong et al., 2026). Moreover, each existing approach is tied to a particular repository artifact, limiting the repositories it can convert into training environments.

We introduce PROGRAMGEN, a repository-to-environment compiler that turns existing repositories into source-hidden, executable reconstruction environments for long-horizon agent training. Given a repository, ProgramGen constructs a workspace, defines the information and execution interfaces available to the agent, and provides a hidden verifier, so that solving the resulting task requires sustained exploration, implementation, execution, and repair rather than recovery of the original source. ProgramGen supports two complementary compilation modes. In PROGRAMGEN-DOC, when adequate repository tests are available, the original tests are retained as the hidden verifier and a test-aligned reconstruction specification is synthesized. In PROGRAMGEN-EXEC, the original program is retained as an isolated, execute-only behavioral oracle (Barr et al., 2014), preventing the agent from directly reading, disassembling, or instrumenting it, while behavioral tests are synthesized to accept the reference and reject degenerate implementations. A common agentic construction pipeline produces validated environments and verifier-selected long-horizon trajectories for training.

We next ask whether ProgramGen supervision teaches capabilities that extend beyond repository reconstruction itself. We fine-tune Qwen3.5-35B-A3B Base (Qwen Team, 2026a) on PROGRAMGEN-DOC, PROGRAMGEN-EXEC, and their mixtures at matched sizes, and evaluate on NL2Repo-Bench (Ding et al., 2025), ProgramBench (Yang et al., 2026b), SWE-bench Verified (Jimenez et al., 2024; Chowdhury et al., 2024), and SWE-bench Multilingual (Yang et al., 2026a). Because the base checkpoint has not undergone agent post-training, improvements in this setting provide a controlled measure of the long-horizon software-engineering behavior induced by reconstruction supervision. We additionally fine-tune the post-trained Qwen3.6-35B-A3B (Qwen Team, 2026b) to test whether the same supervision remains useful once strong agent capabilities are already present.

ProgramGen supervision induces substantial whole-repository, long-horizon capability in the base model. With only 1,000 PROGRAMGEN-DOC trajectories, Qwen3.5-35B-A3B Base improves from 0.5% to 23.0% on NL2Repo-Bench, despite receiving no prior agent post-training. The learned behavior also transfers beyond repository reconstruction: SWE-bench Verified rises from 6.8% to 67.4%, and SWE-bench Multilingual from 6.7% to 54.7%. Notably, PROGRAMGEN-DOC is constructed entirely from Python repositories, yet the resulting model transfers strongly to multilingual software repair, suggesting that the supervision captures reusable interaction behavior rather than only the surface form of the reconstruction task. ProgramGen supervision remains useful even after substantial agent post-training: fine-tuning Qwen3.6-35B-A3B on the full 4,000-trajectory corpus further improves NL2Repo-Bench from 28.8% to 31.8% and SWE-bench Verified from 69.6% to 75.4%. Together, these results suggest that whole-repository reconstruction provides effective supervision for long-horizon software-agent behavior, with transfer beyond reconstruction to more conventional software-engineering tasks.

Our contributions are:

- **Scalable reconstruction environments for long-horizon agent training.** We introduce PROGRAMGEN, a repository-to-environment compiler that converts existing repositories into source-hidden, executable reconstruction tasks under two complementary regimes: PROGRAMGEN-DOC retains the original tests as the verifier, while PROGRAMGEN-EXEC retains an isolated executable reference as a behavioral oracle. Agentic construction produces 10,000 validated environments across five programming languages.
- **Verifier-selected long-horizon supervision.** Each compiled environment provides an executable verifier for validating the task and scoring teacher rollouts, yielding 4,000 selected trajectories, 2,000 per mode, with interactions of up to 512K tokens.

- **Long-horizon agent training with downstream transfer.** Training on PROGRAMGEN trajectories substantially improves zero-to-one whole-repository generation on NL2Repo, while the learned behavior transfers to SWE-bench Verified and SWE-bench Multilingual and remains beneficial for an already post-trained model.

2 RELATED WORK

2.1 TRAINING SOFTWARE AGENTS

Tool-using agents are commonly trained on trajectories distilled from stronger teachers or collected through environment interaction (Chen et al., 2023; Zeng et al., 2024; Xi et al., 2024; Su et al., 2025). For software engineering, SWE-Gym (Pan et al., 2024), R2E-Gym (Jain et al., 2025), SWE-smith (Yang et al., 2026a) and SWE-Lego (Tao et al., 2026) construct executable issue-resolution environments from real repositories, and Repo2Run (Hu et al., 2026) automates the environment setup that such pipelines depend on. These environments target localized repair verified by existing or synthesized unit tests. PROGRAMGEN instead constructs from-scratch whole-repository reconstruction environments that elicit extended exploration, implementation, execution, and repair.

2.2 WHOLE-REPOSITORY RECONSTRUCTION AND LONG-HORIZON TRAINING

Commit0 (Zhao et al., 2025) asks agents to complete libraries with their implementations removed, while NL2Repo-Bench (Ding et al., 2025) evaluates generation of entire repositories from detailed specifications, with both using the original test suites for verification. DeNovoSWE (Zhao et al., 2026) scales documentation-driven whole-repository reconstruction into long-horizon training environments. RepoZero (Zhang et al., 2026) evaluates repository reproduction from API specifications using black-box behavioral verification. ProgramBench (Yang et al., 2026b) removes the dependence on existing tests: agents rebuild a program from its documentation and an execute-only binary and are evaluated with behavioral tests generated from the reference. ProgramDistill (Kim et al., 2026) extends reference-guided software reconstruction to interactive web applications using replayable behavioral traces. MindForge (Chen et al., 2026) constructs source-free training environments around compiled programs and uses long-horizon trajectories to train a coding model.

PROGRAMGEN-DOC follows the test-retaining formulation, whereas PROGRAMGEN-EXEC follows the executable-reference formulation. Relative to ProgramBench, PROGRAMGEN-EXEC additionally isolates the reference in a separate container and exposes it only through an execution stub. Relative to MindForge, PROGRAMGEN uses synthesized behavioral verifiers to score and select teacher trajectories, rather than retaining trajectories based on completion and successful compilation alone. More broadly, PROGRAMGEN constructs verifiable environments across both information regimes and turns them into verifier-selected long-horizon supervision for agent training.

3 METHOD

3.1 ENVIRONMENT COMPILATION FRAMEWORK

PROGRAMGEN compiles a repository into an environment $\mathcal{E} = (W, C, V, O)$, consisting of an isolated workspace W, a contract C describing what to build, a hidden verifier V that scores the final reconstruction, and an optional behavioral oracle O through which the original program can be executed. This separation lets an agent interact with the workspace over extended trajectories, while keeping success anchored to a fixed end-to-end verifier. The two modes instantiate these components differently. PROGRAMGEN-DOC synthesizes the contract while inheriting the repository tests as the verifier, with no oracle. PROGRAMGEN-EXEC instead places the original executable in a separate reference container and exposes it only through an execute-only stub, while synthesizing the verifier from its observed behavior under a minimal contract.

Construction-time access is separated from rollout-time access: *construction agents* may inspect the original repository to build and validate the environment, whereas *rollout agents* see only the workspace, contract, and optional oracle. Environment acceptance depends only on construction-time validation, never on the outcome of a particular rollout. The resulting environment can therefore be reused across agent checkpoints and rollout configurations without changing.

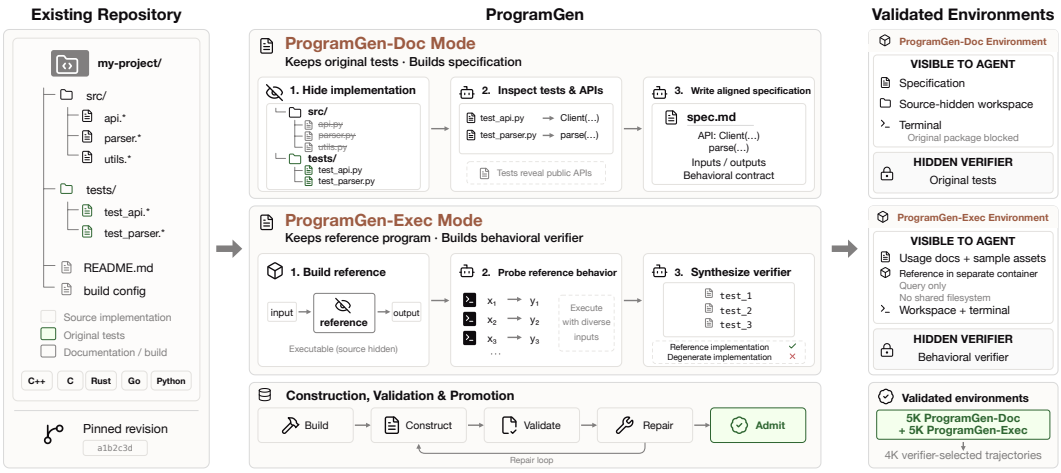

Figure 2: Overview of PROGRAMGEN. PROGRAMGEN-DOC keeps the original tests as the hidden verifier and synthesizes a specification; PROGRAMGEN-EXEC keeps an execute-only reference and synthesizes a behavioral verifier. Candidate environments are validated before acceptance.

Candidates are drawn from an inventory of 200,000 GitHub repositories, 100,000 per mode, filtered by license, activity, and primary-language share, and processed according to a heuristic relevance score (Appendix A.1). This score estimates the quality and usefulness of each repository for environment construction and determines the order in which candidates are processed, but does not affect environment acceptance. For each candidate, PROGRAMGEN checks out a pinned revision and performs a buildability check before synthesis. PROGRAMGEN-DOC requires a runnable test suite that satisfies the test-count, pass-rate, and coverage criteria, while PROGRAMGEN-EXEC requires a runnable and stable reference executable. Only candidates that pass the corresponding check proceed to synthesis.

3.2 PROGRAMGEN-DOC: INHERITED-TEST MODE

The PROGRAMGEN-DOC contract should describe the interfaces and behavior constrained by the inherited tests, so that reconstruction requirements are aligned with evaluation. Following test-aligned documentation (Zhao et al., 2026), PROGRAMGEN combines test execution under coverage with static analysis to identify interfaces referenced by the tests and additional components exercised at runtime. A candidate proceeds only if the test suite contains at least 20 tests, achieves a pass rate of at least 90%, and covers at least 50% of the Python source.

A construction agent organizes the selected components into functional capabilities and drafts their required behavior, usage, and interactions. A critic agent checks each description against repository evidence and requests revision when needed. In parallel, PROGRAMGEN deterministically extracts the interface information required by the tests, including import paths, signatures, constructors, and relevant attributes. These artifacts are assembled into a specification containing a project overview, capability descriptions, an interface reference, and the installation and package conventions needed for reconstruction. Before acceptance, PROGRAMGEN checks that the required components and interfaces are covered and that the specification contains no implementation leakage.

The original tests and fixtures are retained as a hidden verifier. The golden implementation is evaluated in the prepared environment to ensure that the verifier remains valid, after which the implementation, tests, repository history, and other recoverable source artifacts are removed from the rollout workspace. The rollout agent receives only the specification and the prepared runtime workspace. Verifier scoring and additional validation details are provided in Appendix A.2.

3.3 PROGRAMGEN-EXEC: EXECUTABLE-REFERENCE MODE

Reference oracle. In executable-reference reconstruction, the rollout agent may query the original program but must not inspect or recover its implementation (Yang et al., 2026b; Chen et al., 2026). This makes the information boundary part of the environment itself. In our initial rollouts with the reference executable available inside the agent container, the teacher attempted to read, disassemble, or instrument it in 48.8% of trajectories. PROGRAMGEN therefore runs the reference in a separate container and exposes it only through an execute-only stub. The stub forwards arguments, standard input, working directory, and locale to the reference container over a socket and returns standard output, standard error, and the exit status. The two containers do not share a file system, so the rollout agent observes the reference only through these channels.

Test synthesis. A construction agent inspects the source and documentation, probes the reference program, and organizes its behavior into functional categories, such as argument handling, input/output, configuration, and error cases. It then synthesizes tests that check externally observable behavior rather than internal implementation details. Each test is validated against the reference, and PROGRAMGEN additionally checks that it depends only on behavior accessible through the rollout-time oracle. Candidates requiring observations unavailable through this interface are rejected.

Test acceptance. A useful synthesized test should not only pass on the reference implementation, but also reject incorrect or degenerate implementations. Each test is retained only if it passes on the reference in two runs and passes heuristic checks for non-vacuous assertions. Tests passed by any of eight fixed degenerate implementations, such as programs that produce empty output or echo their input, are marked as ignored and excluded from scoring; this filters trivial tests but does not prove that the remaining tests are complete. Reference failures or incomplete reference runs block acceptance. Validation failures are returned to the construction agent with feedback, which revises the tests within at most four attempts, including the initial generation. An environment is accepted when at least 200 active tests and 8 behavior categories survive and the validation checks pass. The verifier score is the number of passed active tests divided by the fixed number of active tests in the evaluation catalog, with incomplete executions unable to receive full credit (Appendix A.2).

Contract. The contract C consists of a fixed task template, identical across tasks except for the source repository, revision, and implementation language it names, together with usage documentation stripped of implementation detail and sample input assets. Container isolation and network restriction prevent retrieving the implementation, but do not rule out recall of a project the teacher has seen during pretraining; we regard this as one of the limitations.

3.4 VALIDATION AND ACCEPTANCE

Both modes pass through the same outer loop: a construction session produces candidate artifacts, and quality evaluation requests targeted repair with concrete failure feedback and the validated build state, or rejects the candidate. Three families of checks apply. *Build* checks establish the runtime and reference described above. *Leakage* checks screen agent-facing materials for source fragments, verify that the workspace contains no recoverable repository artifacts, and keep the verifier payload outside the workspace. *Verifier quality* checks are the mode-specific acceptance conditions in Sections 3.2 and 3.3. Repositories appearing in evaluation benchmarks are excluded at inventory time, and official benchmark test suites are never used as synthesis inputs. The pipeline yields 10,000 validated environments, 5,000 from PROGRAMGEN-DOC and 5,000 from PROGRAMGEN-EXEC.

3.5 TRAJECTORY COLLECTION AND CURATION

Rollouts. We roll out DeepSeek V4.1 Flash (DeepSeek-AI, 2026) through OpenHands SDK 1.33 (Wang et al., 2025) with a 512K-token context, at most 500 iterations, at temperature 0.7 with high reasoning effort. Outbound network access is restricted to the model gateway. A rollout is valid if it terminates by agent decision with a finite verifier score and an intact interaction record; infrastructure failures are discarded. One valid trajectory is kept per environment.

Audit and masking. Valid trajectories are audited for prohibited actions: recovering the implementation from bytecode, packages, or repository history; reading, copying, or instrumenting the reference; reading hidden tests; and network retrieval. The audit is heuristic and distinguishes attempted from confirmed violations; trajectories with confirmed violations are excluded. Within retained trajectories, assistant turns containing malformed tool use, prohibited attempts, or repeated nonproductive errors are assigned zero loss weight while remaining in context. Informative failures, such as a compilation error that the agent then diagnoses, retain supervision: a failed action is training signal when the response to it is.

Selection. The audited trajectories are ranked within each mode by verifier score, and the highest-scoring examples are selected. Partially successful reconstructions remain eligible as long as they pass the audit. We construct nested corpora of 500, 1,000, and 2,000 trajectories per mode, such that each larger corpus contains the smaller one and therefore gradually admits lower-scoring trajectories. Mixed corpora combine PROGRAMGEN-DOC and PROGRAMGEN-EXEC under equal quotas.

3.6 LONG-CONTEXT SUPERVISED FINE-TUNING

Each trajectory is serialized in the student chat format with its full history and trained with full-parameter SFT at a context limit of 524,288 tokens, obtained by extending the native 262,144-token context with YaRN at a scaling factor of 2 (Peng et al., 2024). Only assistant tokens in unmasked turns, including reasoning and tool calls, are prediction targets; system instructions, user messages, and tool observations provide context only. With $m_{i,t} \in \{0, 1\}$ marking supervised tokens,

$$\mathcal{L}_{\mathrm{SFT}}(\theta) = -\frac{1}{Z} \sum_i \sum_t m_{i,t} \log p_\theta(x_{i,t} \mid x_{i,<t}), \qquad Z = \sum_i \sum_t m_{i,t}. \tag{1}$$

Student initialization, data budgets, and optimizer settings are given in Section 4.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Training configurations. Our controlled experiments start from Qwen3.5-35B-A3B Base and vary only the source and amount of PROGRAMGEN supervision. All models are fine-tuned with full-parameter SFT using LLaMA-Factory (Zheng et al., 2024) with the Megatron-LM backend (Shoeybi et al., 2019), using the same training recipe across the controlled Base-model experiments. We train four configurations: 1,000 PROGRAMGEN-DOC trajectories, 1,000 PROGRAMGEN-EXEC trajectories, a size-matched mixture of 500 PROGRAMGEN-DOC + 500 PROGRAMGEN-EXEC trajectories, and a larger mixture of 1,000 + 1,000 trajectories. The first three use the same total budget of 1,000 trajectories, allowing us to compare the training value of the two reconstruction regimes and their combination without confounding data volume. Comparing the 500+500 and 1,000+1,000 mixtures then measures the effect of increasing the amount of mixed supervision. The trajectory sets are nested within each mode: larger corpora retain all examples from the corresponding smaller corpus and add progressively lower-ranked trajectories according to verifier score. Thus, increasing the data budget changes both corpus size and the range of reconstruction quality represented in training.

Student training. Our controlled experiments use full-parameter SFT of Qwen3.5-35B-A3B Base with the same initialization and training recipe across PROGRAMGEN-DOC, PROGRAMGEN-EXEC, and mixed corpora, so that only the training data changes. We retain complete long-horizon trajectories up to 524,288 tokens (512K) rather than splitting them into shorter segments. We separately fine-tune the post-trained Qwen3.6-35B-A3B checkpoint to test whether the same supervision provides additional gains after post-training. Full training configurations are reported in Appendix A.3.

Evaluation. We evaluate on NL2Repo-Bench (Ding et al., 2025), ProgramBench (Yang et al., 2026b), SWE-bench Verified (Jimenez et al., 2024; Chowdhury et al., 2024), and SWE-bench Multilingual (Yang et al., 2026a). For SWE-bench Verified and Multilingual, we allow up to 500 agent steps, a two-hour wall-clock limit, and a 64K-token context window, and report resolved-task rate. NL2Repo-Bench is evaluated with OpenHands 1.33 using up to 500 steps, an eight-hour wall-clock

Table 1: Characteristics of PROGRAMGEN environments and selected training tasks. Corpus size is reported over retained validated environments; all other statistics are computed over the selected 2,000 tasks per mode. Repository size and verifier statistics are medians.

	PROGRAMGEN-DOC	PROGRAMGEN-EXEC
Validated environments	5,000	5,000
Selected training tasks	2,000	2,000
Primary repository languages	Python	Rust, Go, C, C++
Source lines (P50)	2,337	12,707
Source files (P50)	17	53
Verifier tests (P50)	61	274
Source-line coverage (P50)	88.3%	–
Envs. with dummy-pass tests removed	–	32.5%

Table 2: Overall characteristics of the selected training trajectories (2,000 trajectories per setting). Shell density is computed per trajectory as shell executions per 100k tokens.

Metric	PROGRAMGEN-DOC					PROGRAMGEN-EXEC				
	Mean	P50	P90	P95	P99	Mean	P50	P90	P95	P99
Tokens (k)	162.0	157.3	254.1	281.7	324.6	252.6	255.8	392.9	428.7	492.8
Tool calls	85.6	79.0	141.1	164.0	200.0	190.1	188.0	298.0	332.0	393.0
Shell exec.	49.1	46.0	78.0	89.0	121.0	130.6	128.0	202.0	222.0	263.0
Workspace edits	26.3	19.0	56.0	71.0	97.0	40.2	34.0	83.1	103.0	142.0
Shell exec./100k	31.5	30.4	43.4	47.7	55.1	54.2	52.7	68.9	75.1	98.7
Pre-edit shell exec.	13.9	12.0	24.0	30.0	41.0	62.5	61.0	100.0	112.0	134.0
Edit-exec. cycles	6.0	5.0	14.0	17.0	25.0	7.4	6.0	16.0	21.0	32.0
Verifier score	0.778	0.840	1.000	1.000	1.000	0.812	0.887	1.000	1.000	1.000

limit, and the model's native context window; we report both average test pass rate and the fraction of fully solved repositories. For ProgramBench, we allow up to 1,000 model calls, a six-hour wall-clock limit, and a 128K-token context window, and report Almost and Pass Rate. ProgramBench results marked with ‡ use the adapted evaluation protocol described in Appendix A.4.

4.2 ENVIRONMENT CHARACTERISTICS

Table 1 summarizes the validated environments and the tasks selected for training. The two modes cover substantially different repository regimes. Selected PROGRAMGEN-DOC tasks are Python projects with a median of 2.3k source lines, while PROGRAMGEN-EXEC primarily covers compiled-language repositories that are substantially larger, with a median of 12.7k source lines. Their verifiers are also substantial: PROGRAMGEN-DOC retains a median of 61 tests covering 88.3% of source lines, while PROGRAMGEN-EXEC retains a median of 274 active behavioral tests. Degenerate-program filtering removes at least one synthesized test in 32.5% of selected PROGRAMGEN-EXEC environments, showing that reference-passing alone would retain a nontrivial number of weak tests.

4.3 TEACHER TRAJECTORY ANALYSIS

Tables 2 and 3 summarize the 2,000 selected trajectories from each mode. Because the two modes operate on different repositories and language distributions, we interpret these comparisons descriptively rather than causally. The involved definitions can be found in Appendix A.5.

PROGRAMGEN-EXEC spends more context on execution and probing. PROGRAMGEN-EXEC trajectories are substantially longer than PROGRAMGEN-DOC trajectories (252.6k versus 162.0k tokens on average), but their edit density is nearly identical: 15.9 versus 16.2 workspace edits per 100k tokens. The difference is instead concentrated in execution. PROGRAMGEN-EXEC averages 54.2 shell executions per 100k tokens compared with 31.5 for PROGRAMGEN-DOC, and this gap persists across trajectory-length quantiles. Pre-edit execution differs even more strongly:

Table 3: Behavior conditioned on trajectory token length. Quantile thresholds are computed separately for each setting; all values except N are group means. Shell density is averaged over per-trajectory densities, and cycle counts follow Table 2.

Length group	N	PROGRAMGEN-DOC				PROGRAMGEN-EXEC			
		Tokens	Shell/100k	Cycles	Verifier	Tokens	Shell/100k	Cycles	Verifier
$\leq P50$	1,000	105.7k	34.0	2.52	0.872	165.0k	58.0	3.80	0.812
$(P50, P90]$	800	200.6k	29.4	8.20	0.697	315.7k	51.1	9.69	0.808
$(P90, P99]$	180	282.1k	27.5	14.38	0.633	430.6k	47.7	15.54	0.819
$> P99$	20	350.1k	27.7	18.05	0.696	507.2k	45.4	19.10	0.868

PROGRAMGEN-EXEC performs 62.5 shell executions before the first edit on average, compared with 13.9 for PROGRAMGEN-DOC. In contrast, edit-execute-edit cycles increase only modestly, from 6.0 to 7.4 per trajectory. These patterns suggest that much of the additional PROGRAMGEN-EXEC context is spent investigating the reference behavior before and during implementation.

Trajectory length has different associations with success. For PROGRAMGEN-DOC, verifier score decreases with trajectory length, from 0.872 in the shorter half to 0.633 in the $(P90, P99]$ group. For PROGRAMGEN-EXEC, verifier scores remain approximately stable across the same within-mode quantile groups (0.812, 0.808, and 0.819). Thus, long PROGRAMGEN-DOC trajectories are more strongly associated with unsuccessful reconstruction, whereas long PROGRAMGEN-EXEC trajectories are not simply lower-quality failures. Together with the heavier execution and pre-edit probing above, this is consistent with PROGRAMGEN-EXEC using additional context for behavioral discovery. The $> P99$ groups contain only 20 trajectories each and are not interpreted. Because trajectories are selected by verifier score, all relationships are conditional on the retained training corpus.

4.4 MAIN RESULTS

Whole-repository learning from a base model. Table 4 shows that PROGRAMGEN supervision can induce substantial whole-repository capability from a base checkpoint with little prior agent behavior. Qwen3.5-35B-A3B Base passes only 0.5% of NL2Repo instances, whereas fine-tuning on only 1,000 PROGRAMGEN-DOC trajectories raises the pass rate to 23.0%. Under the same evaluation scaffold, this exceeds DeNovoSWE-35B-A3B on NL2Repo (19.1%) while using substantially fewer training trajectories than DeNovoSWE (over 10k). The PROGRAMGEN-EXEC regime also provides a strong learning signal for executable-reference reconstruction, reaching 20.0% average score on ProgramBench under our adapted protocol, compared with 0.7% for the Qwen3.5 base checkpoint under the common evaluation setting. Beyond whole-repository reconstruction, the learned behavior transfers strongly to software repair: SWE-bench Verified increases from 6.8% to 67.4%, and SWE-bench Multilingual from 6.7% to 54.7%. Since PROGRAMGEN-DOC environments are predominantly Python, the latter result provides evidence that the induced agent behavior transfers across both task format and programming language.

Effect of reconstruction mode and data composition. The two modes provide complementary supervision, but their training value is not identical. With the total trajectory count fixed at 1,000, a balanced 500 PROGRAMGEN-DOC + 500 PROGRAMGEN-EXEC mixture reaches 23.2% on NL2Repo, close to the 23.0% obtained from 1,000 PROGRAMGEN-DOC trajectories, while PROGRAMGEN-DOC alone transfers more strongly to SWE-bench Multilingual (54.7% versus 52.7%). PROGRAMGEN-EXEC supervision produces substantial gains over the base model: 1,000 trajectories raise SWE-bench Verified from 6.8% to 66.8% and SWE-bench Multilingual from 6.7% to 51.3%, and reach 20.0% average score on ProgramBench under our adapted protocol. Increasing the corpus to 1,000 trajectories from each mode gives the strongest Qwen3.5 result overall, reaching 24.8% on NL2Repo, 72.4% on SWE-bench Verified, and 60.7% on SWE-bench Multilingual.

Supervision after agent post-training. PROGRAMGEN supervision remains useful after strong agent post-training. Starting from Qwen3.6-35B-A3B, fine-tuning on the full 4,000-trajectory PRO-

Table 4: Main results on NL2Repo, ProgramBench, SWE-bench Verified, and SWE-bench Multilingual. All values are percentages. Frontier-model results are from official releases or leaderboards and may use different scaffolds; all other results use our common settings, except † results from the original paper. ‡ denotes the adapted protocol (Appendix A.4); "–" denotes not evaluated.

Model	NL2Repo		ProgramBench		SWE-bench Verified	SWE-bench Multilingual
	Pass	Full	Almost	Avg.		
Frontier models (reported)						
Claude Opus 5	75.3	–	37.0	74.7	96.0	89.5
GPT-5.6 Sol	56.8	–	15.5	69.9	96.2	74.1
Gemini 3.8 Flash	–	–	–	–	80.0	–
DeepSeek-V4.1-Flash	64.0	–	20.3	–	–	98.2
Kimi K3	58.0	–	17.5	77.8	93.4	80.8
MiniMax M3	42.1	–	–	–	80.5	–
GLM-5.3	58.0	–	19.0	–	95.4	81.3
Prior repository-training methods						
DeNovoSWE-35B-A3B	19.1	1.0	0.0	20.3	64.2	40.0
MindForge-27B†	23.5	–	2.5	49.5	73.8	67.8
Qwen baselines and ProgramGen						
Qwen3.5-35B-A3B Base	0.5	0.0	0.0	0.7	6.8	6.7
DOC (500) + EXEC (500)	23.2	1.9	–	–	69.8	52.7
DOC (1k)	23.0	2.9	–	–	67.4	54.7
EXEC (1k)	–	–	0.0‡	20.0‡	66.8	51.3
DOC (1k) + EXEC (1k)	24.8	2.9	–	–	72.4	**60.7**
Qwen3.6-35B-A3B	28.8	1.9	0.0	**25.7**	69.6	56.7
DOC (2k) + EXEC (2k)	**31.8**	**2.9**	0.0‡	24.2‡	**75.4**	59.3

GRAMGEN corpus improves NL2Repo from 28.8% to 31.8%, SWE-bench Verified from 69.6% to 75.4%, and SWE-bench Multilingual from 56.7% to 59.3%. The gains are smaller than those obtained from the base checkpoint, as expected given the much stronger starting model, but they show that whole-repository reconstruction supervision provides additional value beyond existing post-training. ProgramBench is an exception: the average score is 24.2% under the adapted protocol, compared with 25.7% for the original Qwen3.6 model.

Failure modes. PROGRAMGEN-trained models exhibit a clear trade-off between solution quality and rollout completion. They explore for longer and are more likely to exhaust the interaction or context budget before leaving a valid submission. However, conditional on rollouts that terminate normally with a valid submission, their success rate is higher than that of the Qwen3.6-35B-A3B baseline. This conditional comparison is not a replacement for the overall benchmark score, since termination is itself part of agent performance, but it indicates that a major remaining bottleneck is completion and submission discipline rather than implementation quality alone.

5 CONCLUSION

We introduced PROGRAMGEN, a scalable environment-construction framework that turns existing repositories into source-hidden, verifiable reconstruction tasks for long-horizon agent training. Its two complementary modes, PROGRAMGEN-DOC and PROGRAMGEN-EXEC retain either the original tests or an executable reference, enabling environment construction under different information regimes while preserving source isolation and end-to-end verification. Training on PROGRAMGEN trajectories substantially improves zero-to-one whole-repository generation from a base model, transfers to conventional software engineering tasks, and provides further gains after agent post-training. Our results also show that the value of long-horizon supervision depends on how the underlying environment is constructed, not merely on the number of successful trajectories. More broadly, PROGRAMGEN demonstrates that existing software repositories can be systematically converted into scalable, verifiable supervision for long-horizon agents.

AI USE STATEMENT

In this work, we used generative AI tools for generating synthetic training data and other artifacts used by our environment-construction pipeline, including specifications, agent trajectories, and behavioral verification artifacts. We also used generative AI tools to provide feedback on research methodology and experimental design.

Additionally, we used generative AI tools for modifying scientific figures, suggest experimental parameters, and summarize or analyze existing literature. We have reviewed all AI-assisted work. In particular, AI-generated training data and artifacts were subjected to the build, execution, and validation modes described in the paper. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. Swe-bench+: Enhanced coding benchmark for llms. *arXiv preprint arXiv:2410.06992*, 2024.

Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. Fireact: Toward language agent fine-tuning. *arXiv preprint arXiv:2310.05915*, 2023.

Yihao Chen, Shi Chang, Khaled Chawa, Feng Lin, Boyuan Chen, Shaowei Wang, and Ahmed E Hassan. Mindforge: Teaching small language models whole-life-cycle software engineering via source-free program synthesis. *arXiv preprint arXiv:2607.27146*, 2026.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench verified, 2024. URL `https://openai.com/index/introducing-swe-bench-verified/`.

DeepSeek-AI. Deepseek-v4.1-flash: Pushing the limits of kv cache compression, 2026.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, et al. Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*, 2025.

Ruida Hu, Chao Peng, Junjielong Xu, and Cuiyun Gao. Repo2run: Automated building executable environment for code repository at scale. *Advances in Neural Information Processing Systems*, 38:32679–32718, 2026.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164*, 2025.

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 *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Jacob Kahn, Felix Kreuk, Yuxiang Wei, Jannik Kossen, and Emily McMilin. Repo state loopholes during agentic evaluation. *GitHub issue, SWE-bench/SWE-bench*, 2025.

Jeonghye Kim, Minseon Kim, Young Jin Kim, Matheus Pereira, Marc-Alexandre Côté, Alessandro Sordoni, Xingdi Yuan, and Zhengyan Shi. Programdistill: From interactive web apps to verifiable reference-guided swe tasks. *arXiv preprint arXiv:2609.18805*, 2026.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*, 2024.

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. Yarn: Efficient context window extension of large language models. In *International Conference on Learning Representations*, volume 2024, pp. 31932–31951, 2024.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026a. URL https://qwen.ai/blog?id=qwen3.5.

Qwen Team. Qwen3.6-35B-A3B: Agentic coding power, now open to all, April 2026b. URL https://qwen.ai/blog?id=qwen3.6-35b-a3b.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Hongjin Su, Ruoxi Sun, Jinsung Yoon, Pengcheng Yin, Tao Yu, and Sercan Arik. Learn-by-interact: A data-centric framework for self-adaptive agents in realistic environments. In *International Conference on Learning Representations*, volume 2025, pp. 89044–89097, 2025.

Chaofan Tao, Jierun Chen, Yuxin Jiang, Kaiqi Kou, Shaowei Wang, Ruoyu Wang, Xiaohui Li, Sidi Yang, Yiming Du, Jianbo Dai, Zhiming Mao, Xinyu Wang, Lifeng Shang, and Haoli Bai. Swe-lego: Pushing the limits of supervised fine-tuning for software issue resolving. *CoRR*, abs/2601.01426, 2026. doi: 10.48550/ARXIV.2601.01426. URL https://doi.org/10.48550/arXiv.2601.01426.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Z Xi, Y Ding, W Chen, B Hong, H Guo, J Wang, D Yang, C Liao, X Guo, W He, et al. Agentgym: Evolving large language model-based agents across diverse environments. arxiv 2024. *arXiv preprint arXiv:2406.04151*, 2024.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026a.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, et al. Programbench: Can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, 2026b.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. Agenttuning: Enabling generalized agent abilities for llms. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 3053–3077, 2024.

Zhaoxi Zhang, Yiming Xu, Jiahui Liang, Weikang Li, Xiaoshuai Chen, Liwei Qian, Xin Pei, Jizhou Huang, Run Sun, and Yunfang Wu. Repozero: Can llms generate a code repository from scratch? *arXiv preprint arXiv:2605.07122*, 2026.

Jiale Zhao, Guoxin Chen, Fanzhe Meng, Wayne Xin Zhao, Ruihua Song, Ji-Rong Wen, and Kai Jia. Denovoswe: Scaling long-horizon environments for generating entire repositories from scratch. *arXiv preprint arXiv:2606.10728*, 2026.

Wenting Zhao, Nan Jiang, Celine Lee, Justin Chiu, Claire Cardie, Matthias Gallé, and Alexander Rush. Commit0: Library generation from scratch. In *International Conference on Learning Representations*, volume 2025, pp. 12061–12076, 2025.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand, 2024. Association for Computational Linguistics. URL http://arxiv.org/abs/2403.13372.

Ziqian Zhong, Aditi Raghunathan, and Nicholas Carlini. Impossiblebench: Measuring llms' propensity of exploiting test cases. In *International Conference on Learning Representations*, volume 2026, pp. 124959–124986, 2026.

A Appendix

A.1 Heuristic Repository Relevance Score

We rank candidate repositories using a deterministic score combining GitHub metadata and repository-tree features. The weights are fixed heuristics; buildability and task quality are verified separately.

Features and normalization. Let $[x]_0^1 = \min(1, \max(0, x))$. Count features use logarithmic saturation, $S(n; \tau) = [\log(1+n)/\log(1+\tau)]_0^1$. Size features use a log-space triangular function $T(x; l, p, u)$, which is zero outside $[l, u]$, rises linearly in $\log x$ from zero at l to one at p, and falls to zero at u.

The metadata features are: star maturity $S(s; 1000)$; repository-size fit $T(k; 50, 3000, 30000)$, with k in KB; activity $[(y - 2020)/6]_0^1$, where y is the last-push year; primary-language share; metadata presence $0.65\mathbf{1}[\text{description}]+0.35\mathbf{1}[\text{topics}]$; and procedure-specific keyword relevance. Invalid push timestamps receive zero activity.

Tree features count test files, documentation files, CI files, build manifests, CLI source paths, and distinct source-root groups, with saturation thresholds of $40, 20, 5, 5, 8, 4$, respectively. Counts use filename and directory patterns, such as `tests/`, `test_*.py`, `docs/`, `README*`, `.github/workflows/`, `Cargo.toml`, `pyproject.toml`, and `cmd/`. CLI paths also include source files with stems `cli`, `main`, or `__main__`. Source-root groups use the first path component, or `src/<component>` for nested source layouts. Tree-size fit is $T(\widehat{\text{LOC}}; 500, 25000, 500000)$, where $\widehat{\text{LOC}}$ is source-file bytes divided by 40 and rounded.

Keyword relevance. We tokenize the lowercased repository name, description, and topics, preserving hyphenated words. For ProgramGen-Exec, the keyword set comprises `analyzer`, `cli`, `command-line`, `compiler`, `converter`, `decoder`, `encoder`, `formatter`, `generator`, `interpreter`, `linter`, `parser`, `serializer`, `terminal`, and `validator`. The keyword score is $\max\{\min(1, 0.2u + 0.3v), 0.65I\}$, where I indicates any match in this set, and u, v count distinct text-token and exact-topic matches in the same set augmented with `devtool`. For ProgramGen-Doc, the score is 0.75 when any of `api`, `framework`, `library`, `package`, `plugin`, `sdk`, or `toolkit` matches, and 0.25 otherwise.

Score aggregation. Table 5 gives both stages of the weighted score. Let $\mathbf{f}_q$ contain the metadata features and $\mathbf{g}$ contain the tree features. We compute

$$C_q = \max(0, \mathbf{w}_q^\top \mathbf{f}_q - 0.12L), \tag{2}$$

$$E_q = \alpha_q C_q + \mathbf{v}_q^\top \mathbf{g}, \tag{3}$$

$$Q_q = \text{round}_6\left([\gamma \max(0, E_q - 0.03D)]_0^1\right). \tag{4}$$

Here L indicates a match to any low-signal keyword: `awesome`, `boilerplate`, `collection`, `demo`, `examples`, `learning`, `template`, or `tutorial`. The penalty applies once. D indicates a truncated tree response, and γ is 1 for supported runtime languages and 0.1 otherwise. Without tree features, the score falls back to $\text{round}_6([\gamma C_q]_0^1)$; the configured full-enrichment selection requires tree features.

Candidates are selected in descending score order within language quotas. Ties prefer enriched records, then use a deterministic SHA-256 ordering with seed `20260818`. These structural signals prioritize candidates but do not establish executable tests, coverage, or a runnable reference program.

Feature	PROGRAMGEN-EXEC	PROGRAMGEN-DOC
Metadata-stage weights $\mathbf{w}_q$		
Star maturity	0.18	0.18
Repository-size fit	0.19	0.22
Activity	0.13	0.12
Language share	0.12	0.16
Metadata presence	0.08	0.10
Keyword relevance	0.30	0.22
Final-stage weights $(\alpha_q, \mathbf{v}_q)$		
Metadata score C_q	0.18	0.16
Test files	0.17	0.28
Documentation files	0.09	0.12
CI files	0.09	0.09
Build manifests	0.09	0.10
Tree-size fit	0.14	0.12
CLI paths	0.24	0.00
Source-root groups	0.00	0.13

Table 5: Fixed weights for repository relevance ranking.

A.2 PROGRAMGEN VERIFIER SCORING

PROGRAMGEN-EXEC scoring. The verifier matches test results against a fixed catalog of active tests. The score is P/A, where P is the number of matched passing tests and A is the catalog size. Ignored tests are excluded, and missing tests receive no credit. Execution is complete only when all branches are processed, none times out, and all active tests have matching results. A nominal full score is reset to zero if execution is incomplete.

PROGRAMGEN-DOC scoring. The verifier parses test output, prioritizing per-test pytest results and falling back to supported runner summaries. Per-test results are deduplicated by test identifier, allowing partial credit even when a timeout prevents a final summary. Let E be the expected test count recorded during construction, and let P, F, and R denote observed passes, failures, and errors. The score is

$$s = \frac{P}{\max(E, P + F + R)},$$

with zero assigned when the denominator is zero. Skipped tests do not contribute to the observed count. Execution is complete only when results are parsed, the observed count reaches E, the exit status is consistent with the reported outcomes, and no recognized compilation failure is detected. Incomplete runs retain partial credit, but a nominal full score is reset to zero. Failures before the test phase or unparseable test output receive zero.

A.3 TRAINING CONFIGURATION

We perform full-parameter supervised fine-tuning using LLaMA-Factory with the Megatron backend. Both Base and Instruct models use a global batch size of 64, a maximum sequence length of 524,288 tokens, BF16 precision, weight decay of 0.01, gradient clipping at 1.0, and a random seed of 42. We train Base models for 5 epochs with a peak learning rate of 5×10^{-5} and 10% linear warmup followed by cosine decay. Instruct models are trained for 1 epoch with a learning rate of 1×10^{-5} and cosine decay without warmup. Training examples are shuffled and sorted by length within each global batch; sequence packing is disabled.

Base. We fine-tune Qwen3.5-35B-A3B-Base for 5 epochs with a peak learning rate of 5×10^{-5}.

Instruct. We fine-tune Qwen3.6-35B-A3B for 1 epoch with a peak learning rate of 1×10^{-5}.

A.4 PROGRAMBENCH EVALUATION PROTOCOL

ProgramGen models use a reinforced prompt that encourages maintaining a buildable and submitted workspace throughout the rollout, together with a context-budget reminder once context usage

exceeds 60%. For chat templates that preserve prior reasoning, the reminder is inserted as a new user message; otherwise, it is appended to the latest tool observation to avoid discarding previous reasoning. ProgramGen models use a 524,288-token context window, while the Qwen3.6-Instruct baseline uses the original prompt without context-budget reminder. All runs use the same OpenHands SDK 1.33 adapter, with at most 1,000 model calls, a six-hour wall-clock limit, temperature 1.0, top-p 0.95, and top-k 20. The adapter also issues the same end-of-budget reminder when fewer than 20 calls or 10 minutes remain. These settings describe our evaluation adapter rather than the official ProgramBench harness.

A.5 Trajectory Metric Definitions

Trajectory statistics are computed from the recorded interaction histories. Token counts use the Qwen3.6 Instruct tokenizer and the training-format serialization, counting each conversation once rather than repeated context prefixes across model requests. Unless otherwise stated, statistics are computed per trajectory and summarized with equal trajectory weight.

Shell and edit statistics. A shell execution is a non-empty terminal command-dispatch call accepted by the tool interface; polling, control inputs, and malformed or rejected calls are excluded. Each qualifying tool call counts once, regardless of the number of commands it contains. For trajectory i, shell density is $100{,}000S_i/T_i$, where S_i and T_i are its shell-execution count and token count, respectively.

Workspace edits are statically recognized file-editor or shell write actions. For source-file statistics, we retain files with recognized programming-language extensions while excluding identifiable tests, probes, checks, fuzzers, and validation artifacts. File-editor actions require a recognized successful edit; shell-based edits additionally require a recognized write target and successful outer-shell completion.

Pre-edit exploration. Pre-edit shell executions are completed shell calls whose observations become available before the first recognized successful source-file edit. Calls issued in the same assistant turn as the first edit, or whose completion is not known before that turn, are excluded. Trajectories without a recognized source-file edit are omitted from this statistic (4 ProgramGen-Doc and 21 ProgramGen-Exec trajectories). This metric captures pre-edit interaction, which may include reference probing, workspace inspection, dependency queries, and setup rather than investigation alone.

Edit–execute–edit cycles. We count a cycle when a recognized source-file edit is followed by an observed execution of the candidate implementation and then by a subsequent source-file edit. Consecutive edits without an intervening execution are treated as one edit phase, so multiple executions between two edit phases contribute at most one cycle. Cycle detection excludes loss-masked actions and shell calls that simultaneously contain recognized workspace writes, avoiding ambiguous within-call ordering. The intermediate execution need not fail, and the subsequent edit need not repair an error; the metric is therefore only a conservative proxy for iterative execution and revision.

Candidate executions are identified through static command analysis, including invocations of workspace executables, common test runners and language-toolchain run/test commands, Python modules associated with the task or edited workspace, and interpreter calls to recognized workspace or verification scripts. Inline and previously recorded helper scripts are inspected statically when possible; trajectory code is never executed by the analysis. Pure build and inspection commands are excluded unless they also match a candidate-execution pattern. The same recognition mode is applied to both settings.

Additional analysis details. The primary shell, edit, and pre-edit statistics include loss-masked actions. As a sensitivity check, excluding loss-masked shell calls reduces the mean pre-edit counts to 12.62 for ProgramGen-Doc and 57.09 for ProgramGen-Exec, leaving the qualitative difference unchanged.

For Table 3, token-length quantiles are computed separately for ProgramGen-Doc and ProgramGen-Exec, and statistics are averaged within $\leq P50$, $(P50, P90]$, $(P90, P99]$, and

$> P99$ groups. These groups represent relative positions within each setting rather than matched absolute length ranges.

The final verifier score is the reward produced by the corresponding PROGRAMGEN verifier: repository tests for PROGRAMGEN-DOC and synthesized behavioral tests validated against the reference implementation for PROGRAMGEN-EXEC. Because the two settings use different task sets and verifier semantics, these scores are not interpreted as a controlled cross-setting comparison.