SERA: Soft-Verified Efficient Repository Agents

Ethan Shen[1,2]   Daniel Tormoen[1]   Saurabh Shah[1]   Ali Farhadi[1,2]   Tim Dettmers[1,3]

[1]Allen Institute for AI   [2]University of Washington   [3]Carnegie Mellon University

Abstract

✦Ai2

Open-weight coding agents should hold a fundamental advantage over closed-source systems: they can be specialized to private codebases, encoding repository-specific information directly in their weights. Yet the cost and complexity of training has kept this advantage theoretical. We show it is now practical. We present Soft-Verified Efficient Repository Agents (SERA), an efficient method for training coding agents that enables the rapid and cheap creation of agents specialized to private codebases. Using only supervised finetuning (SFT), SERA achieves state-of-the-art results among fully open-source (open data, method, code) models while matching the performance of open-weight models like Devstral-Small-2. Creating SERA models is 26x cheaper than reinforcement learning and 57x cheaper than previous synthetic data methods to reach equivalent performance. Our method, Soft Verified Generation (SVG), generates thousands of trajectories from a single code repository. Combined with the cost-efficiency, this enables specialization to private codebases. SVG is built on two observations that emerged from simplification of previous methods: First, soft verification, where instead of testing the correctness of synthetic coding data via unit tests, we only compare the partial line-by-line overlap of patches generated from two rollouts. This removes the need for test infrastructure and enables data generation from any repository, practically removing limits on the amount of data we can generate from a single codebase as well as what codebases can be used. Second, vague instructions can diversify training data, increasing the proportion of data focused on non-bug related changes like refactoring. We find that these vague instructions improve SWE-bench performance as well as bug-focused data. In more detail, SVG is based on two rollouts from an agent: in the first, a teacher model is prompted with a vague instruction to make a change to a codebase starting from a randomly selected function, producing a trajectory and patch. This trajectory is converted into a synthetic pull request. In the second, the teacher model attempts to reproduce the patch given only the pull request description. Soft verification compares the two patches using line-level recall for training data selection. Taken together, this creates a cheap pipeline for high-quality data that enables rapid experimentation. We show through power scaling curves that private codebase specialization is highly sample efficient and matches or exceeds teacher model performance at low costs. Beyond repository specialization, we apply SVG to a larger corpus of codebases, generating over 200,000 synthetic trajectories. We use this dataset to provide detailed analysis of scaling laws, ablations, and confounding factors for training coding agents. Overall, we believe our work will greatly accelerate research on open coding agents and showcase the advantage of open-source models that can specialize to private codebases. We release SERA as the first model in Ai2's Open Coding Agents series, along with all our code, data, and Claude Code integration to support the research community.
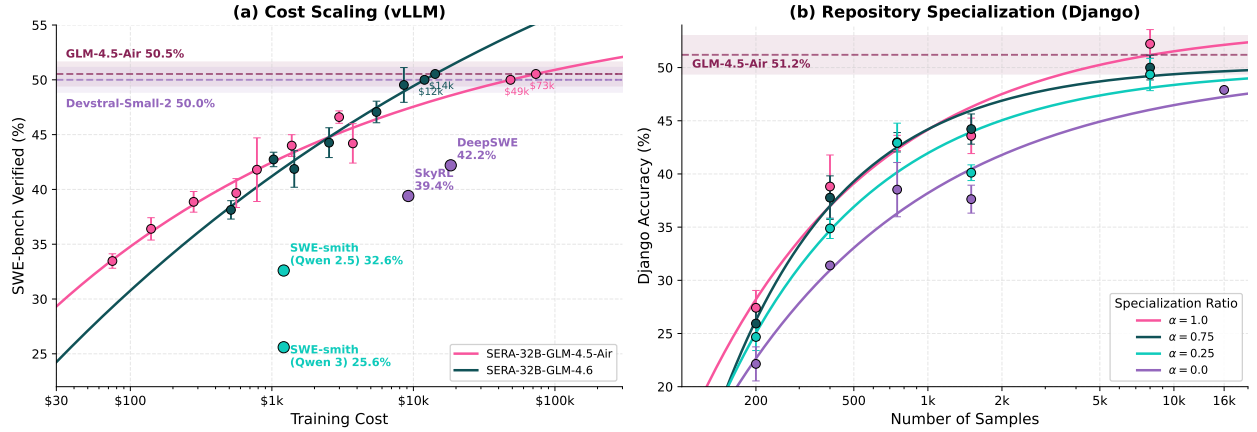
Figure 1 **(a)** Scaling and cost comparison of coding agent training approaches using self-hosted vLLM inference. **(b)** Repository specialization scaling law on Django, where α denotes the fraction of Django-specific data in the training mixture. With full specialization ($\alpha = 1.0$), the model matches teacher performance at 8k samples; general data alone ($\alpha = 0.0$) requires 25k samples – a $3.5\times$ advantage in sample efficiency.

1 Introduction

Coding agents have become central to software development and are increasingly applied to tasks beyond traditional engineering. While, closed-source coding agents are more powerful, open-weight models should hold a fundamental advantage in many applications because they can be specialized to private codebases, allowing them to learn repository-specific patterns, conventions, and domain knowledge. Despite this clear opportunity, the cost and complexity of training open-weight coding agents has kept this advantage theoretical. In this work, we show it is now practical.

As the first release in Ai2's Open Coding Agents series, our method trains a 32B coding agent with simple supervised finetuning achieving state-of-the-art open-source results at 40 GPU days ($2000) or match strong open-weight models like Devstral-Small-2 at a budget of $9000. When specializing to a particular codebase, our pipeline can match or exceed teacher model performance at $1300.

Training coding agents traditionally requires either reinforcement learning or complex synthetic data pipelines, both demanding resources beyond what most teams can provide. Reinforcement learning requires sandboxed execution environments, distributed training infrastructure, and rollout orchestration. The complexity of this infrastructure is reflected in team sizes that average 12 or more authors in recent work Cao et al. (2025); Luo et al. (2025); Wei et al. (2025); Da et al. (2025). Synthetic data approaches like SWE-smith (Yang et al., 2025) require setting up test environments, generating valid bugs, and verifying bugs through test suites. These barriers have concentrated coding agent development in well-resourced industry labs and larger teams at academic institutions. Starting from limited compute and a small team (32 GPUs, 3 researchers), we prioritized reducing experimentation costs, which led us to systematically strip away pipeline complexity and discover which components actually matter for training effective coding agents.

We found that much of the complexity in prior pipelines is unnecessary. Firstly, soft verification, where patches are checked by partial line-by-line matching rather than executed test suites, produces training data of equal quality to full test-based verification. At the scales we test, the degree of verification has minimal effect on downstream performance, removing the need for test infrastructure entirely and enabling data generation from any repository. This makes synthetic data generation much more straightforward. Secondly, some coding instructions are inherently vague, and we observe that models prompted to fix these issues often produce changes such as refactoring or documentation improvements that are more representative of real world tasks than just bug fixes. Rather than requiring bug-focused data, we find that this general coding data is equally effective to improve performance. Together, these findings mean that generating effective training data requires neither test infrastructure nor complex bug-injection pipelines. Additionally, unlike other methods, how much data we can generate from repositories are not limited by test coverage or quality.
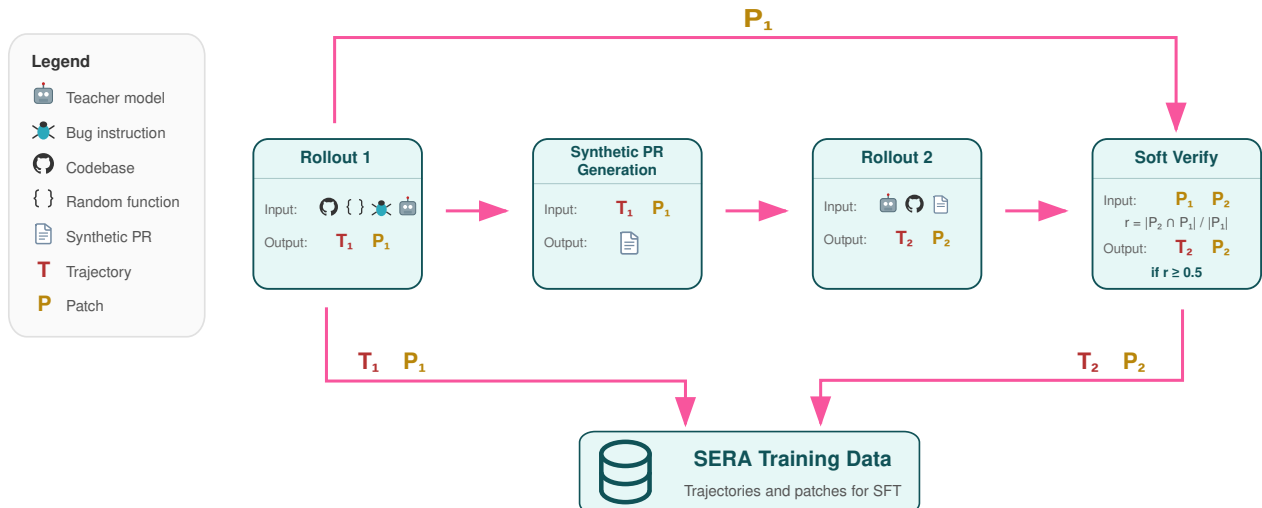
Figure 2 Overview of SVG (Soft Verified Generation). In the first rollout, a teacher model is prompted to make a change starting from a randomly selected function, producing a trajectory and patch. This trajectory is converted into a synthetic pull request. In the second rollout, the teacher attempts to reproduce the patch given only the PR description. Soft verification compares the two patches using line-level recall for training data selection. We use $r \geq 0.5$ as an example threshold.

The resulting cost reduction and data abundance make repository specialization practical. We show that open-weight models specialized to a codebase can match or exceed the performance of the teacher model used to generate their training data. This is intuitive: a student model with repository-specific knowledge encoded in its weights can outperform a teacher that accesses the same codebase only through its context window. The advantage this creates extends beyond privacy. Even organizations willing to share their code would need to wait many months until the next training run of a frontier model includes their data. And while LoRA adapter options for frontier models exist these are often to impractical or costly for large-scale deployments. Open-weight specialization allows practitioners to generate data from their repositories, fine-tune, and deploy immediately, iterating as their codebase evolves. At low cost, any team can build and deploy a small specialized model that outperforms frontier systems on their own codebase – an advantage that grows with the codebase and that frontier labs cannot close regardless of their scale.

We introduce **SERA** (Soft-verified Efficient Repository Agents), a 32B coding agent that achieves 49.5%/54.2% on SWE-bench Verified when evaluated at 32K/64K context, state-of-the-art for fully open-source models. We exceed the performance of previous open-source solutions with a total cost of \$2,000 for both data generation and training (40 GPU days). Our method, **SVG** (Soft Verified Generation; see Figure 2 for an overview), achieves equivalent performance to SkyRL at $26\times$ lower cost and to SWE-smith at $57\times$ lower cost when self-hosting inference via vLLM. Using the z.ai API, these advantages increase to $53\times$ and $115\times$ respectively. These efficiency factors are derived from our scaling laws that capture both per-sample savings and data quality gains (see Appendix D for detailed cost breakdowns). Effectively specializing to a single repository requires approximately 8,000 trajectories (\$1,300). These trajectories are generated from randomly selected functions and contain no information about evaluation issues or their solutions. We validate our findings across multiple seeds and use scaling laws as robustness checks, adopting a methodology designed to ensure reported effects reflect genuine signal rather than noise. To further support open coding agent research, we provide extensive analyses covering ablations on data quality factors, model-specific pitfalls, and common confounding factors that have slowed progress in this area.

We release **SERA** as the first model in Ai2's Open Coding Agents series, along with all our code, 200,000 synthetic coding agent trajectories, and Claude Code integration.

2 Background

This section introduces the core concepts and prior approaches for training coding agents. We cover the standard evaluation benchmark, the structure of agent systems and their training data, and the two main paradigms for training on synthetic data: data generation and reinforcement learning.

2.1 SWE-bench

SWE-bench (Jimenez et al., 2023) is the standard benchmark for evaluating coding agents on real-world software engineering tasks. While other benchmarks exist that are more comprehensive, such as Terminal-Bench (Merrill et al., 2026), SWE-bench offers broader comparisons between methods and models, and its confounding factors are better understood. Each task is derived from a real GitHub issue and pull request from 12 popular Python repositories such as Django, Sympy, and Sphinx. Given an issue description, the agent must produce a patch that resolves the issue. The repository's test suite is run before and after applying the patch and a task is considered solved if previously failing tests now pass and no previously passing tests are broken. SWE-bench Verified is a curated subset where human annotators have verified that each task is solvable and that the tests correctly validate the solution. We use SWE-bench Verified for all evaluations in this work.

2.2 Agent Scaffolds and Training Data

Coding agents operate through scaffolds that define the tools available to the agent and how it interacts with the environment. SWE-agent (Yang et al., 2024) is a widely used scaffold that provides tools for viewing files, editing code, and executing bash commands. The agent receives observations from these tools and produces actions in a loop until the agent decides to perform the last action, which in this case of SWE-bench is the submission of the final software patch.

A **rollout** is one complete execution of the agent on a task, from receiving the issue to submitting a solution. The sequence of actions, observations, and reasoning produced during a rollout is called a **trajectory**. A **patch** is the final output: a line-by-line diff specifying additions and deletions to the codebase.

Trajectories are the training data for coding agents. The two main approaches for generating trajectories are synthetic data generation and reinforcement learning, which we describe in the following subsections.

Two practical considerations affect both approaches. First, coding agent trajectories can be very long, often requiring 32K tokens or more of context (Yang et al., 2024). This makes experimentation slow and expensive, and means that models must handle long contexts well and efficiently to be effective coding agents. Second, coding agents rely heavily on tool calling: if a model cannot reliably follow the tool format and produce valid tool calls, it cannot function as a coding agent regardless of its other capabilities.

2.3 Synthetic Data Generation

Synthetic data generation creates trajectories by having a strong teacher model solve synthetic tasks—artificially constructed problems designed to mimic real issues but generated programmatically rather than from actual user reports—then using those trajectories to train a smaller student model. This teacher-student distillation approach separates data generation from training, allowing each to be optimized independently.

The standard approach, exemplified by SWE-smith (Yang et al., 2025), generates training data through bug injection. Starting with a repository that has a passing test suite, the pipeline programmatically injects bugs that cause tests to fail, generates an issue description from the bug, has the teacher model solve the issue, and verifies correctness by checking that tests pass again. This requires test infrastructure, valid bug generation, and execution environments for verification.

The cost structure of this approach is significant. Using Sonnet 3.7, each SWE-smith trajectory costs approximately $0.52 including issue creation and rollout (see Section D for a detailed cost comparison). Beyond direct costs, the complexity of CPU-based test execution adds complex infrastructure and slows iteration on experiments.

A limitation of teacher-student distillation is that the student's performance is largely bounded by the teacher's capability. While there are cases where students can slightly exceed their teachers (Hinton et al., 2015; Furlanello et al., 2018)—and we see similar results in our work (see Sections 4.2 and 5.3)—the gains are modest. This means that at the frontier, where no stronger teacher exists, synthetic data generation may not be sufficient for further progress and reinforcement learning might be required.

2.4   Reinforcement Learning

Reinforcement learning trains coding agents by having them generate rollouts and learning from reward signals based on whether tasks are solved. Unlike synthetic data generation, the model being trained is also the model generating trajectories.

This has both advantages and disadvantages. The disadvantage is that if the model is initially too weak, improvement is slow or nonexistent because it generates mostly unsuccessful trajectories to learn from. The advantage is that a strong model can continue to improve through self-play, since it is not bounded by a separate teacher's capabilities. At the very frontier of model performance, reinforcement learning may be the only path to further progress.

However, reinforcement learning introduces substantial infrastructure complexity. It requires online rollouts during training, sandboxed execution environments, and distributed systems for coordinating rollouts with gradient updates. This complexity is reflected in team sizes: recent RL papers for software engineering agents average 12 or more authors, including SWE-RL with 9 authors (Wei et al., 2025), SkyRL-Agent with 15 authors (Cao et al., 2025), and SWE-rebench with 9 authors (Badertdinov et al., 2025). Beyond team size, reinforcement learning is known to be highly unstable and difficult to use in practice, with training runs sensitive to hyperparameters, reward shaping, and random seeds (Henderson et al., 2018; Engstrom et al., 2020).

For small academic teams like ours, with one researcher, two engineers, and two advisors, reinforcement learning approaches are difficult to execute. This motivated our focus on supervised methods that achieve comparable results with simpler infrastructure.

2.5   Verification

Verification determines whether a generated trajectory is suitable for training. Traditional approaches use **unit test verification**: the patch must pass all relevant tests, confirming that the synthetic bug was correctly resolved. This ensures correctness but limits data generation to repositories with comprehensive test coverage and sufficient test quality.

Our method introduces **soft verification**: instead of executing tests, we compare the generated patch against a reference patch using line-level recall. If the generated patch contains most or all of the lines from the reference patch, we consider it verified. This removes the need for test infrastructure and enables data generation from any repository. We describe the details of our soft verification approach in the following section.

3   Method

3.1   Soft Verified Generation (SVG)

The key intuition behind **SVG** is that clear errors in syntax, logic, and failing unit tests are only a subset of real world coding pull requests (PRs). Indeed, it is extremely common for PRs to be more obscure, aimed at refactoring code, enforcing style requirements, or tweaking behavior. This section will be an overview of **SVG** and then describe each of the components in more detail. Figure 2 visualizes every step of **SVG**.

In **SVG**, we rethink the criteria that define a valid synthetic PR. While traditional synthetic approaches explicitly focus synthetic issues on failed unit tests to ensure samples represent correct code, we instead broaden the definition of a PR to include any instruction that attempts to create *some* desired change in a codebase **C**. This interpretation is central to our approach. The key insight is that a trajectory's value for training lies not in producing a fully correct patch, but in the skills it demonstrates, for example, how to

interpret an instruction, navigate a codebase, and translate intent into code.

SVG is composed of two rollouts. We use $\mathbf{T}$ and $\mathbf{P}$ to denote the trajectory and patch created by a rollout. In SVG, we use a teacher model $\mathbf{M}$ to generate rollouts. In the first rollout, we prompt $\mathbf{M}$ with a random function $\mathbf{func_i}$ from codebase $\mathbf{C}$ and a bug prompt $\mathbf{bug_j}$ sampled from a set of 51 bug types $\mathbf{B}$. This produces trajectory $\mathbf{T_1}$ and patch $\mathbf{P_1}$. We then convert $\mathbf{T_1}$ into a synthetic PR $\mathbf{synth\_PR}$ using a demonstration PR $\mathbf{PR}$ sampled from SWE-Bench Verified. In the second rollout, $\mathbf{M}$ is prompted with $\mathbf{synth\_PR}$ and tasked to reproduce the original change, producing trajectory $\mathbf{T_2}$ and patch $\mathbf{P_2}$. Soft-verification compares $\mathbf{P_2}$ against $\mathbf{P_1}$ using line-level recall r. The combination of these steps is **SVG**. We provide a general mathematical overview below and more detail in the following sections.

$$\text{First Rollout:} \quad \mathbf{T_1}, \mathbf{P_1} = \mathbf{M}(\mathbf{func_i}, \mathbf{bug_j}, \mathbf{C}) \tag{1}$$

$$\text{Synthetic PR Generation:} \quad \mathbf{synth\_PR} = \mathbf{M}(\mathbf{T_1}, \mathbf{PR}) \tag{2}$$

$$\text{Second Rollout:} \quad \mathbf{T_2}, \mathbf{P_2} = \mathbf{M}(\mathbf{synth\_PR}, \mathbf{C}) \tag{3}$$

$$\text{Soft Verification:} \quad r = \frac{|\mathbf{P_2} \cap \mathbf{P_1}|}{|\mathbf{P_1}|} \tag{4}$$

If $r = 1$, the trajectory is hard-verified; if $0 < r < 1$, soft-verified; if $r = 0$, unverified.

We will now explain each component in depth.

Agent Workflow: We use SWE-agent (Yang et al., 2024) to generate trajectory rollouts. SWE-agent allows users to define a variety of tools available to an agent and gives users the ability to adjust settings such as the length of tool outputs and context history. To reduce the effect of confounding factors, we use SWE-agent in its vanilla state: we only provide the agent with the ability to run a view tool, edit tool, submission tool, and bash commands. Furthermore, we do not truncate context history or tool outputs at any point during rollouts. While truncations are frequently done to avoid context window errors, we noticed that many previous works use slightly different heuristics, making it difficult to objectively compare performance. Additionally, we believe an important trait of coding agents is their ability to solve tasks while avoiding unnecessarily long tool calls and outputs. However, for transparency, we ablate different heuristic choices in Appendix ??.

First Rollout: At a high-level, the first rollout works as follows: We prompt the teacher model $\mathbf{M}$ with "There is a $\mathbf{bug_j}$ downstream of function $\mathbf{func_i}$.", where $\mathbf{bug_j}$ is a high-level description of a bug type, $\mathbf{func_i}$ is a randomly chosen function in the codebase $\mathbf{C}$. The function $\mathbf{func_i}$ serves as an arbitrary starting point for the agent. We run the pipeline once for every function in the codebase. Each $\mathbf{bug_j}$ is randomly sampled from a larger list of 51 types of bugs $\mathbf{B}$ and asks the model to fix issues ranging from state management to code clarity. We generate this list of 51 bugs from papers that study bug distributions in software systems (Just et al., 2014; Widyasari et al., 2020). We intentionally leave the prompt vague to widen the range of acceptable changes. We rollout for a maximum of 115 steps, although this limit is rarely reached.

Occasionally, the model is unable to find a bug that both aligns with the prompt and is related to the provided function. To handle this edge case, we separately ask the teacher model $\mathbf{M}$ to self-evaluate its fix after the rollout is finished. We accept the trajectory $\mathbf{T_1}$ unless the teacher model decides it did not make a change aligned with the prompt. In that case, we reject the trajectory and we perform another rollout with a different sampled prompt until a valid change is produced or a limit of three runs is reached. About 2% of rollouts are rejected during the first rollout, and less than 1% fail all three rollouts. We also discard trajectories that produce duplicate patches, although we observe that this is extremely rare. The final patches $\mathbf{P_1}$ from accepted trajectories are saved as ground truth. Importantly, these synthetic tasks are generated without reference to evaluation benchmarks—no information about the bugs or GitHub issues in SWE-bench Verified is contained in our training data.

$$\mathbf{T_1}, \mathbf{P_1} = \mathbf{M}(\mathbf{func_i}, \mathbf{bug_j}, \mathbf{C}), \tag{5}$$

Synthetic PR: Next, to create $\mathbf{synth\_PR}$ to guide the second stage, we provide the teacher model $\mathbf{M}$ with its first rollout $\mathbf{T_1}$, which contains relevant reproduction scripts, execution traces, and the final software patch. Similar to SWE-smith (Yang et al., 2025), we also include a demonstration PR $\mathbf{PR}$ sampled from SWE-Bench

Verified (Jimenez et al., 2023). The teacher is then asked to write a new PR that follows the format of the demonstration PR.

$$\text{synth\_PR} = \mathbf{M}(\mathbf{T_1}, \mathbf{PR}),\tag{6}$$

Second Rollout: In the second rollout, we only use the synthetic PR **synth_PR** as the input, with the goal of reproducing the initial patch. The trajectory $\mathbf{T_2}$ is again capped at 115 steps, and the resulting patch $\mathbf{P_2}$ is saved.

$$\mathbf{T_2}, \mathbf{P_2} = \mathbf{M}(\text{synth\_PR}, \mathbf{C}),\tag{7}$$

Soft Verification: We evaluate the second rollout patch $\mathbf{P_2}$ using recall against the first rollout patch $\mathbf{P_1}$ by assessing edits at a line-by-line granularity. If $\mathbf{P_2}$ contains every change from $\mathbf{P_1}$, then the recall is $r = 1$ and the second rollout is considered hard-verified. If $0 < r < 1$, then the rollout is considered soft-verified. Finally, if $r = 0$, then it is considered unverified.

$$r = \frac{|\mathbf{P_2} \cap \mathbf{P_1}|}{|\mathbf{P_1}|},\tag{8}$$

where r is the line-level recall.

Setup Details: We use a suite of 121 codebases for data generation, which are a subset of the 128 codebases released by SWE-smith (Yang et al., 2025).[1] Each codebase $\mathbf{C}$ is encapsulated inside of a docker container. We use GLM-4.5-Air (GLM-4.5 Team et al., 2025) as our teacher model $\mathbf{M}$ for all experiments unless otherwise specified. GLM-4.5-Air has several advantages: it allows for scaled experiments, has full reasoning traces, is powerful, and is easy to deploy on commonly available GPUs. Pairing **SVG**'s generation efficiency and GLM-4.5-Air's cost efficiency makes a robust scientific investigation of coding agent scaling possible.

Indeed, current data generation strategies for coding agents are often bottlenecked by their reliance on closed-source models as teachers, whose API costs make studies of coding agent scaling impractical and hamper statistical reliability, since repeating evaluations across different random seeds becomes very costly. Furthermore, closed-source providers often hide full reasoning traces, which are essential for data quality (Section 5.4) and APIs are prone to change or adjust model quality depending on demand. As an open-weight model, GLM-4.5-Air can be run locally, avoiding these issues. GLM-4.5-Air also strikes a powerful balance between performance and model size: it provides Claude 3.7 Sonnet[2] level performance while being fully deployable on 8 H100s, or 4 H100s at a lower context length, or 2 H100s if quantized. We hope that this will significantly reduce barriers for practitioners and researchers who want to train, use, and study coding agents at scale.

3.2 Training

We use Qwen 3-32B (Team et al., 2025b) as our primary base model over models like Qwen 2.5 (Team et al., 2024) due to Qwen 3-32B's stronger tool calling performance, which better reflects the improving capabilities of current and future base models. This mirrors similar choices from recent work on coding agents (Sonwane et al., 2025b; Cao et al., 2025; Luo et al., 2025). We fully fine-tune up to Qwen 3's native context length of 32768 and train our models for 3 epochs using a learning rate of `1e-5` and weight decay of 0.01. We primarily use axolotl (Axolotl maintainers and contributors, 2023) for training and vLLM (Kwon et al., 2023) for model hosting.

We prioritize training on trajectories that are ≤ 32768 tokens in length. To increase sample size as needed, we selectively truncate longer trajectories based on the ratio of trajectory steps within the context limit—we term this "truncation ratio". In Section 5.2, we explore the effects of truncation in depth and explain why it must be done with caution.

Method	Open Source			Base Model	Teacher	Context	Resolve Rate
	Code	Model	Data				
SkyRL-8B	✓	✓		Qwen 3-8B	—	32K	9.4%
Nex-N1-8B	✓	✓		InternLM3-8B	—	32K	20.3%
SERA-8B-GA (Ours)	✓	✓	✓	Qwen 3-8B	GLM-4.5-Air	32K	31.7% ± 0.4%
SERA-8B (Ours)	✓	✓	✓	Qwen 3-8B	GLM-4.6	32K	31.7% ± 0.9%
Qwen 3-32B	✗	✓	✗	32B	—	32K	24.4%
SWE-smith				Qwen 3-32B	Claude 3.7	32K	25.6% ± 1.1%
SWE-smith	✓	✓	✓	Qwen 2.5-32B	Claude 3.7	32K	32.6%
FrogBoss-32B	✗	✓	✗	Qwen 3-32B	Claude 4 Sonnet	32K	35.0%
GLM-4.7-Flash	✗	✓	✗	30B	—	32K	37.3% ± 2.0%
SkyRL-Agent	✓	✓		Qwen 3-32B	—	32K	39.4%
DeepSWE	✓	✓		Qwen 3-32B	—	32K	42.2%
Qwen 3-Coder-30B	✗	✓	✗	30B	—	32K	45.0%
Kimi-dev	✗	✓	✗	72B	—	32K	48.6%
Devstral-Small-2	✗	✓	✗	24B	—	32K	50.0% ± 1.3%
GLM-4.5-Air	✗	✓	✗	110B	—	32K	50.5% ± 1.3%
GLM-4.6	✗	✓	✗	357B	—	32K	60.8%
SERA-32B-GA (Ours)	✓	✓	✓	Qwen 3-32B	GLM-4.5-Air	32K	46.6% ± 0.7%
SERA-32B (Ours)	✓	✓	✓	Qwen 3-32B	GLM-4.6	32K	49.5% ± 1.9%
GLM-4.7-Flash	✗	✓	✗	30B	—	64K	39.7% ± 1.8%
SWE-Swiss	✓	✓		Qwen 2.5-32B	—	128K	45.0%
Qwen 3-Coder-30B	✗	✓	✗	30B	—	256K	51.6%
CWM	✗	✓	✗	32B	—	128K	53.9%
FrogBoss-32B	✗	✓	✗	Qwen 3-32B	Claude 4 Sonnet	64K	54.6%
GLM-4.5-Air	✗	✓	✗	110B	—	64K	57.4% ± 0.5%
Devstral-Small-2	✗	✓	✗	24B	—	64K	59.1% ± 1.1%
GLM-4.7-Flash	✗	✓	✗	30B	—	128K	59.2%
Devstral-Small-2	✗	✓	✗	24B	—	256K	68.0%
SERA-32B-GA (Ours)	✓	✓	✓	Qwen 3-32B	GLM-4.5-Air	64K	51.7% ± 1.1%
SERA-32B (Ours)	✓	✓	✓	Qwen 3-32B	GLM-4.6	64K	54.2% ± 1.4%

Table 1 SWE-bench Verified performance comparing **SERA** against other coding agent training methods. We separate by sequence length as this is the largest confounding factor. Gray rows are open-weight models, white rows are fully open-source models. Standard deviations reported where available from our replications using 3 random seeds. Nex-N1-8B from Team et al. (2025a), InternLM3-8B from Cai et al. (2024), CWM from Copet et al. (2025).

4 Main Results

We primarily evaluate on SWE-bench Verified (Jimenez et al., 2023), a curated subset of SWE-bench where human annotators have verified that each task is solvable and that the tests correctly validate the solution. In this section, we focus on three evaluation settings: (1) a head-to-head comparison that controls for the teacher model while comparing against other synthetic data methods, (2) a scaling law study that examines how our approach scales with data size and predicts when we reach certain performance thresholds, and (3) a focused benchmarking of how well our approach can target specific codebases for improved performance.

---

[1]We exclude 7 codebases that contain little to no Python code, such as repositories consisting of a single file or minimal Python content.

[2]https://www.anthropic.com/news/claude-3-7-sonnet

A key consideration in our evaluation methodology is controlling for evaluation context length. Context length has a significant impact on memory footprint, even among models of equal sizes. Doubling the context length often requires increasing memory by nearly the same factor. We also observe that context length is one of the factors that most strongly differentiates model performance—methods evaluated at 64K or 128K context often appear substantially stronger than those evaluated at 32K context, even when the underlying model capabilities are similar. To ensure fair comparisons across deployment configurations, we explicitly report and control for context length in all experiments, and we group results by context size in our leaderboard (Table 1). For every experiment, performance is averaged across three random seeds, a practice we find essential for reliable conclusions given the high variance in coding agent evaluations (see Section 6 for detailed statistical analysis).

4.1 Controlled Comparisons

Method	SWE-smith	SERA	BugPilot	SERA
Base model	Qwen 3-32B	Qwen 3-32B	Qwen 3-32B	Qwen 3-32B
Teacher	Claude 3.7	Claude 3.7	Claude 4 Sonnet	Claude 4 Sonnet
Eval context size	32K	32K	64K	64K
Sample size	4776	4776	5819	5319
SWE-bench Verified	25.27% ± 0.61%	30.00% ± 1.41%	49.87%	48.53% ± 0.31%

Table 2 SWE-Bench Verified results comparing SWE-smith baseline with SERA across different teacher models and context sizes. All experiments use Qwen 3-32B as the base model. Additional baseline comparisons including SWE-smith on Qwen 2.5 and SERA with GLM-4.5-Air are provided in Appendix B.

The goal of this section is to understand the differences between SERA and other synthetic data generation methods when we control for teacher model, verification method, and evaluation context length. Because hard verification rates vary based on repository difficulty and teacher model capability, sample sizes can only be approximately matched—we generate samples and filter post-hoc for hard verification, which introduces some variance in the final dataset sizes.

We also note an important methodological consideration regarding context management during evaluation. Some agent frameworks employ optimizations such as retaining only the last few tool calls in context rather than the full trajectory history. While this compression allows models to appear effective at longer apparent context lengths, it introduces a confounding factor: methods using such optimizations may appear to benefit from increased context without the associated computational cost. Furthermore, such optimizations cause key-value cache invalidation during inference, which is prohibitively expensive for practical deployment. For fair comparison, we evaluate all methods using full context retention without such optimizations, ensuring that reported context lengths accurately reflect the actual information available to the model.

We compare against SWE-smith and BugPilot using hard-verified trajectories from the second rollout. This ensures that our training data distribution mimics that of other synthetic setups (i.e. synthetic issue descriptions and working code). From Table 2, in a head-to-head comparison with the same teacher and sample size, SERA yields better performance trained on hard-verified trajectories. Additional comparisons showing that SWE-smith is optimized for Qwen 2.5 and that GLM-4.5-Air provides substantial improvements as a teacher model are provided in Appendix B.

We also evaluate at 64K context size and compare against BugPilot BaseMix (Sonwane et al., 2025a), the base mixture in BugPilot's training data, which combines real and synthetic issues from R2EGym and SWE-smith using Claude 4 Sonnet as the teacher. Because BugPilot's data is not public, we choose BaseMix because its reported sample size is closest to our largest Claude 4 Sonnet run. Still, our train set contains approximately 10% fewer samples. Despite this, in a head-to-head comparison, our results nearly match BugPilot's performance.

These results demonstrate that the data quality of our approach is high. Even when controlling for the teacher model, SERA matches real and synthetic approaches that use complicated bug generation pipelines and unit test verification.

4.2 Scaling Experiments

SERA significantly simplifies the process of generating massive amounts of coding data by circumventing the need to introduce synthetic bugs into codebases and validate them with unit test execution. We take advantage of this property to generate three large-scale datasets from the codebases described in Section 3, using both GLM-4.5-Air and GLM-4.6 as teachers.

- Sera-4.5A-Lite is generated by running our data generation pipeline once for every function across all 121 codebases using GLM-4.5-Air as the teacher. This results in approximately 36,000 T_1 and 36,000 T_2 trajectories.
- Sera-4.5A-Full is a superset of Sera-4.5A-Lite. We continue our generation from Sera-4.5A-Lite, looping through every $func_i$ up to three total times. Each time, a new bug prompt is sampled for the first rollout. This ensures that every trajectory is unique even for the same $func_i$. We stop generation after several days, reaching a total of 70,000 T_1 and 70,000 T_2 trajectories.
- Sera-4.6-Lite mimics the setup of Sera-4.5A-Lite, but uses GLM-4.6 as the teacher model. We generate another 36,000 T_1 and 36,000 T_2 trajectories for Sera-4.6-Lite.

Combined, our datasets contain over 200,000 trajectories, resulting in the largest open-source dataset for coding agents to date. We separate these trajectories by teacher model and rollout stage. For T_2 trajectories, we further group them by verification threshold, with boundaries at $r = 0$, 0.25, 0.50, 0.75, and 1. We independently scale both T_1 trajectories and T_2 trajectories until a truncation ratio of 0.88 is reached. Because there are multiple verification thresholds for T_2 rollouts, we choose to scale $r = 0$ (completely unverified) trajectories, which has the highest data count. Our decision was influenced by experiments in Section 5.1, which indicate that completely unverified T_2 rollouts are of equal or better quality than any verified rollouts.

Using Sera-4.6-Lite, we train SERA-32B and set a new state-of-the-art on SWE-Bench Verified for fully open-source 32B models evaluated at 32K context, with open-weight models like Devstral-Small-2-24B and larger models such as GLM-4.5-Air well within uncertainty bounds of one standard deviation. Evaluating at 64K context, SERA-32B again sets a state-of-the-art among fully open-source models, matching open-weight models such as FrogBoss-32B and only outperformed by Devstral-Small-2-24B (Rastogi et al., 2025) among models with similar parameter counts. It is important to note that unlike these models, SERA-32B was not trained past 32K tokens and did not use any reinforcement learning, two factors that place it at a disadvantage at longer contexts. Still, SERA-32B performs extremely well and does not appear to have saturated yet.

We also train SERA-32B-GA using Sera-4.5A-Lite. While SERA-32B-GA lags behind SERA-32B, it still outperforms all other fully open-source models at 32K and 64K context lengths. Interestingly, SERA-32B-GA is able to match SERA-32B at low and intermediate sample sizes, after which point SERA-32B-GA's performance saturates. This suggests that the benefits of strong teacher models primarily emerge in high compute regimes. For researchers and practitioners, this means that it may be optimal to use a weaker teacher model depending on final performance goals and overall budget. Figure 3 highlights this crossover point, where the scaling curves for GLM-4.5-Air and GLM-4.6 intersect.

4.3 Repository Specialization

SERA is the first synthetic data generation strategy that operates totally independent of a repository's unit tests. This allows users to rapidly specialize a base model to any downstream codebase, including private repositories. To emulate this scenario, we use SERA to generate data from the three largest repositories in SWE-Bench Verified: Django, Sympy, and Sphinx. Crucially, our synthetic training data is generated entirely independently of the evaluation instances, containing no information about the actual GitHub issues or their solutions in SWE-bench Verified. These repositories represent 231 (46.2%), 75 (15.0%), and 44 (8.8%) of the 500 instances in SWE-Bench Verified, respectively.

Initial Data Generation: Because every SWE-Bench instance is sourced from a unique commit, the set of instances from each repository will span multiple years. To account for this, we identify the earliest commit and latest commit in SWE-Bench Verified from each repository and generate data from five equally spaced commits in that period. While some functions are repeated across commits, each commit presents the
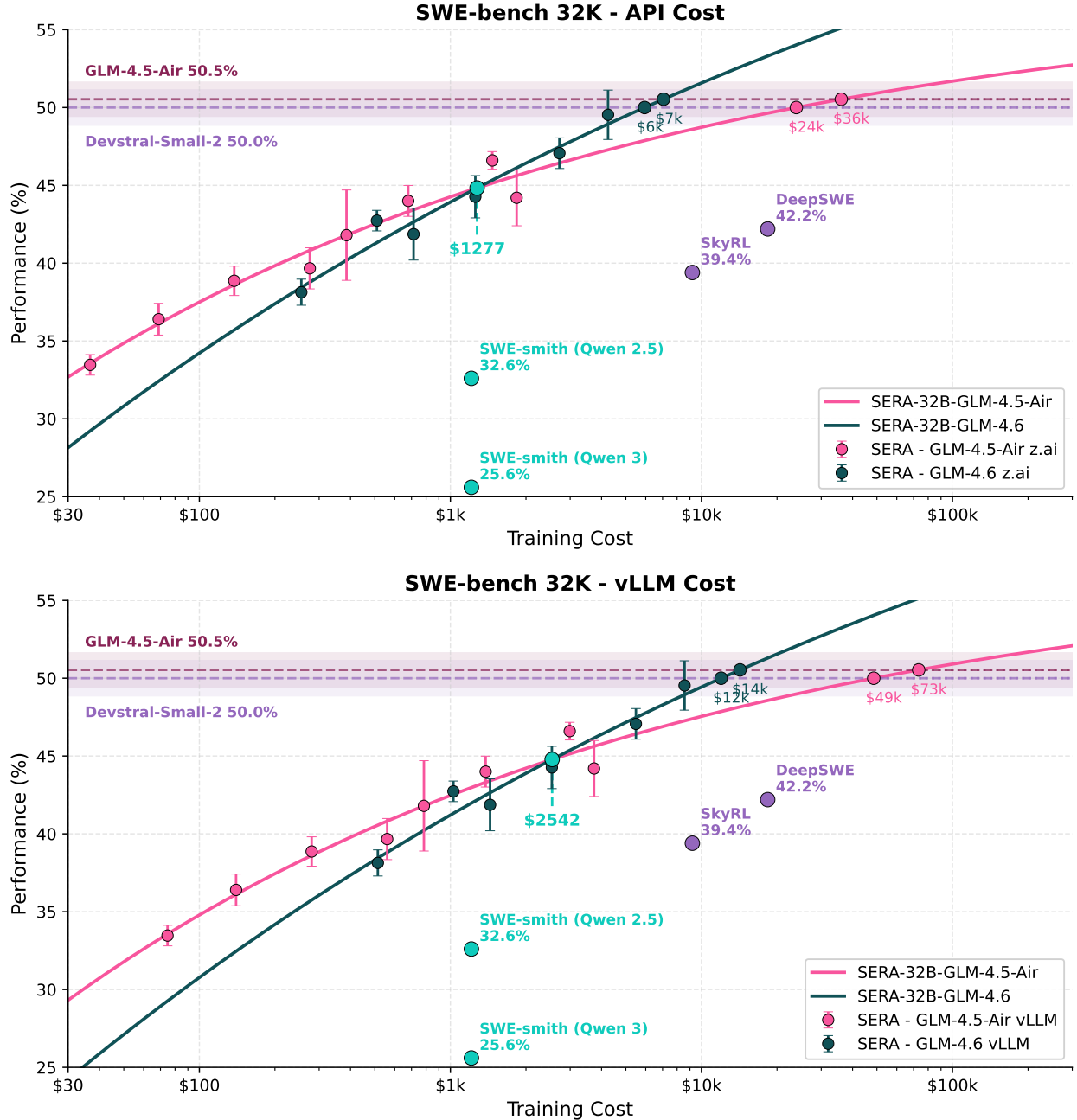
Figure 3 Scaling and cost comparison of coding agent training approaches. **Top:** API cost when using z.ai with cached input pricing. **Bottom:** vLLM cost when self-hosting the teacher model. Horizontal lines indicate the cost at which our scaling law predicts matching Devstral-Small-2 and GLM-4.5-Air performance. Exact data points are provided in Table 11.

codebase in a different context, which ensures trajectories in both the first and second rollout remain unique. Aggregating across commits, we obtain between 46000 and 54000 trajectories for each repository combined across both rollouts. We decide to train on both rollouts to increase sample size since the majority of the generated trajectories exceed 32768 tokens. We investigate the effects of mixing rollouts in Section 5.5. Due to compute constraints, we train on 8000 trajectories per repository rather than the full dataset; however, we release all generated trajectories to enable future research to explore larger-scale specialization.

Data Verification and Filtering: We soft-verify T_2 rollouts with a verification threshold of 0.5. We cap T_1 rollouts based on patch size and observations length. This selects against T_1 rollouts that over-edit or make excessively long tool calls, a tendency that can quickly use up context. We find that this filtering significantly improves specialization performance, which we further investigate in Section 5.3.

Finally, for each repository, we train on 3,000 soft-verified T_2 rollouts and 5,000 filtered T_1 rollouts. We note that these specific proportions were chosen based on preliminary experiments within our compute budget; a more systematic exploration of the optimal mixture would be valuable future work. In this setup, we match or exceed the teacher model GLM-4.5-Air on Django and Sympy instances, and also outperform Devstral-Small-2-24B (SoTA ≤32B parameters), while nearly matching their performance on Sphinx (Table 3). This result is intuitive: the student encodes repository-specific knowledge in its weights, while the teacher can only access the codebase through its context window. We note that at 64K evaluation context, SERA underperforms baselines like Devstral-Small-2 because we train only at 32K context while these models are trained at 64K or longer; see Appendix C for 64K results. These results highlight that given the right data, it is possible to produce and even exceed state-of-the-art performance on specifically targeted repositories.

Model	Django (231)	Sympy (75)	Sphinx (44)
SERA-32B-Django	**52.23% ± 1.64%**	-	-
SERA-32B-Sympy	-	**51.11% ± 1.54%**	-
SERA-32B-Sphinx	-	-	37.14% ± 6.95%
GLM-4.5-Air	51.20% ± 1.80%	48.89% ± 3.08%	**43.51% ± 0.58%**
Devstral-Small-2-24B	51.30% ± 1.72%	47.56% ± 4.68%	38.95% ± 4.24%

Table 3 Specialization results at 32K context comparing GLM-4.5-Air (teacher) and fine-tuned Qwen 3-32B (student) on the three largest repositories in SWE-Bench Verified. Fine-tuned models are trained on 8000 synthetic trajectories from each repository. Results averaged over three seeds. Devstral-Small-2-24B results from Rastogi et al. (2025). See Table 13 in the appendix for 64K evaluation results.

Specialization Scaling Law: To understand how data composition affects specialization efficiency, we fit scaling laws across different mixtures of Django-specific and general coding data (Figure 4). We define the specialization ratio α as the fraction of repository-specific data in the training mixture. At $\alpha = 1.0$ (pure Django data), the model matches teacher performance (GLM-4.5-Air at 51.2%) with only 8,000 samples. In contrast, $\alpha = 0.0$ (pure general data) is unable to reach equivalent performance even at 16,000 samples. Intermediate mixtures ($\alpha = 0.75$, $\alpha = 0.25$) show increasing asymptotic performance as the proportion of specialized data increases. This indicates that when training for a target codebase, the ratio of specialized data is the most important factor. Assumptions of ANOVA were met. A one-way ANOVA revealed a statistically significant effect of specialization ratio at 1,500 samples, $F(3,8) = 10.78$, $p = .003$, $\eta^2 = .80$. Post-hoc comparisons using Tukey's HSD showed significant differences between $\alpha = 1.0$ vs. $\alpha = 0.0$ ($p = .009$) and $\alpha = 0.75$ vs. $\alpha = 0.0$ ($p = .005$). This indicates that repository-specific data yields better performance than general data at equivalent sample sizes.

5   Ablations and Analysis

In this section, we conduct comprehensive data ablations studying design choices in SERA. We focus on the impacts of verification; truncation; specialization; filtering; dataset mixing; and evaluation uncertainty. For these experiments, we draw data from Sera-4.5A-Lite.

5.1   Verification

In Figure 5, we ablate four different verification thresholds for T_2 trajectories: $r = 0.0$, 0.25, 0.75, and 1. We also plot the performance of models trained on T_1 trajectories (which are inherently unverified) at each scale for comparison. We study each verification threshold using only complete trajectories that fit within 32768 tokens, with the final datapoint of each curve representing the largest possible dataset for that threshold.
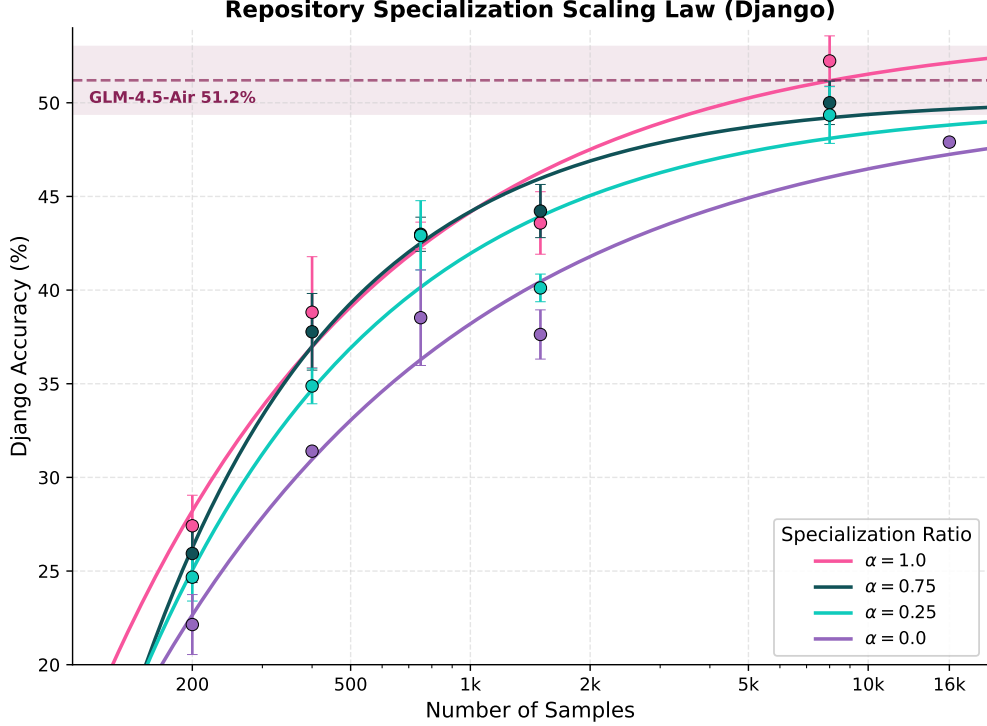
Figure 4 Scaling law for repository specialization on Django. The specialization ratio α denotes the fraction of Django-specific data in the training mixture, with the remainder being general coding data. Dashed horizontal lines show the performance of GLM-4.5-Air and Devstral-Small-2 on Django instances, with shaded regions indicating ± 1 standard deviation. With full specialization ($\alpha = 1.0$), the student model matches teacher performance at approximately 8,000 samples, significantly outperforming training on general data alone ($\alpha = 0.0$). Specialization performance increases with the ratio of Django-specific data.

Only using trajectories ≤ 32768 tokens reduces the total number of trainable trajectories but allows us to avoid confounding factors introduced by truncation.

If verification was essential for performance, we would expect to observe increasing performance as the verification threshold increases. Instead, scaling up to 7400 samples, we find that all verification thresholds perform similarly. For example, at the maximum scale, training on $\mathbf{T_2}$ trajectories that are soft-verified at $r = 0.5$ shows no benefit over training on completely unverified $\mathbf{T_2}$ trajectories. Furthermore, models trained on $\mathbf{T_1}$ trajectories from the first rollout result in similar SWE-Bench Verified performance at each scale, despite representing a completely different distribution of coding tasks.

These results suggest that verification is not a necessity for high quality coding data, a behavior that is similar to what has been observed in other types of reasoning tasks (Chandra et al., 2026). We hypothesize that this is because even incorrect trajectories are can contain important skills, such as how to convert an intention into a relevant code edit even if the intention does not perfectly address the PR at hand. The assumption of normality was violated. A Kruskal-Wallis H-test revealed no statistically significant difference between verification thresholds, $H(3) = 7.19$, $p = .066$, $\varepsilon^2 = .52$. This confirms that soft verification performs as well as hard verification, and even unverified data achieves comparable results.

5.2 Truncation

Truncation is a popular but understudied practice across coding agent research to handle trajectories from teacher models whose context windows are larger than the base model's. Without truncation, base model context limits make it impossible to train on significant amounts of data: 23.26% of SWE-smith and 24.83% of **Sera-4.5A-Lite** (verified at $r = 1$) exceed 32768 tokens.
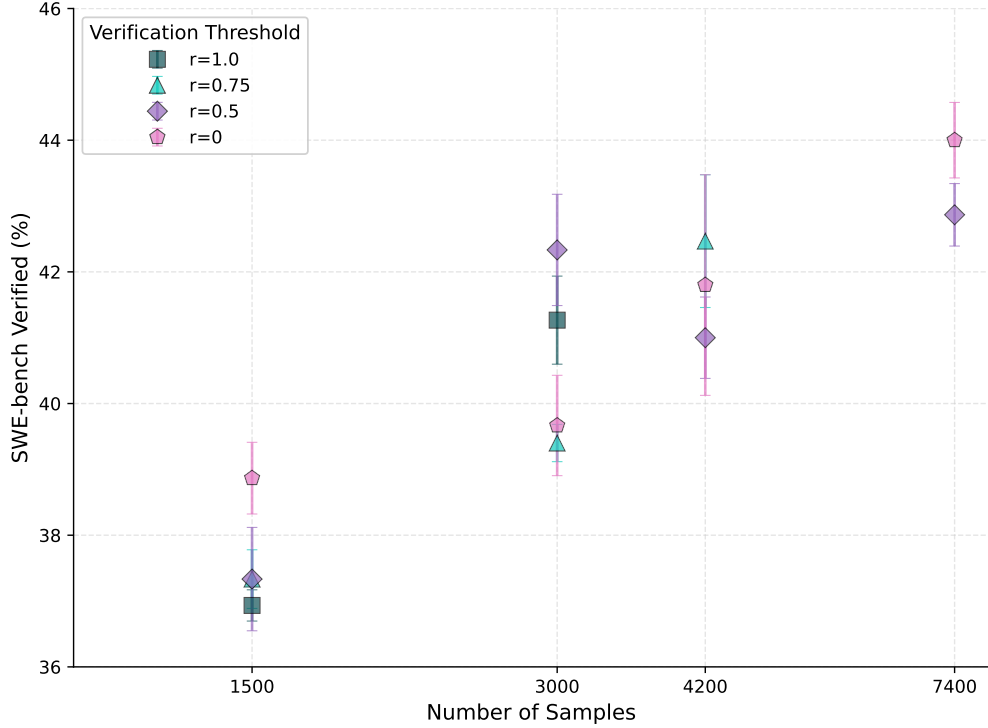
Figure 5 Verification analysis comparing soft and hard verification approaches. Scaling curves show SWE-bench Verified performance for different verification thresholds ($r = 0.0, 0.25, 0.75, 1.0$) on $\mathbf{T_2}$ trajectories alongside unverified $\mathbf{T_1}$ trajectories. All thresholds achieve similar performance at each scale, indicating that strict verification provides no significant benefit over soft or even unverified data.

As a result, current SFT methods will slice long trajectories to fit inside the base model's context window. While this allows every data sample to be used, it assumes that all sliced trajectories are similar quality. We hypothesize that this is a faulty assumption. For example, a trajectory that represents only 50% of a trajectory's total steps is intuitively more noisy than a sliced trajectory that represents 95% of a trajectory's total steps.

To test this hypothesis, we order $\mathbf{T_1}$ trajectories from **Sera-4.5A-Lite** based on the ratio of trajectory steps that fit in 32768 tokens, a property we term "truncation ratio". We partition the ordered $\mathbf{T_1}$ trajectories into subsets of 3000 samples each. This forces each subsequent partition to contain trajectories with strictly lower truncation ratios than the previous partition. We then train Qwen3-32B on each partition. $\mathbf{T_1}$ trajectories work well because they are longer than $\mathbf{T_2}$ trajectories on average while exhibiting similar scaling trends. This allows us to study the effect of training on a wide range of truncation ratios with a non-trivial amount of data and expect findings to translate.

In Figure 6 we plot SWE-Bench Verified performance against the average truncation ratio from every partition. Surprisingly, we find that the best data comes from trajectories that have high truncation ratios but are not fully contained in 32768 tokens. Subsequent truncation ratios result in gradually decreasing performance. We suspect that this is due to a combination of factors, such as longer trajectories reflecting more difficult tasks and that a model's final steps are typically focused on the redundant task of submitting its solution instead of problem solving.

We further explore this phenomenon in Table 4, where we compare the effect of selecting for high truncation ratios against the current practice of using sliced trajectories arbitrarily. We find that randomly picking sliced trajectories results in much lower performance than curating for high truncation ratios. This suggests that existing truncation approaches are suboptimal and may hurt performance. These results inform our scaling experiments in Section 4.2, where we extend our datasets by first ordering by truncation ratio and then truncating until a ratio of 0.88 is reached. We conservatively pick this threshold to avoid deterioration
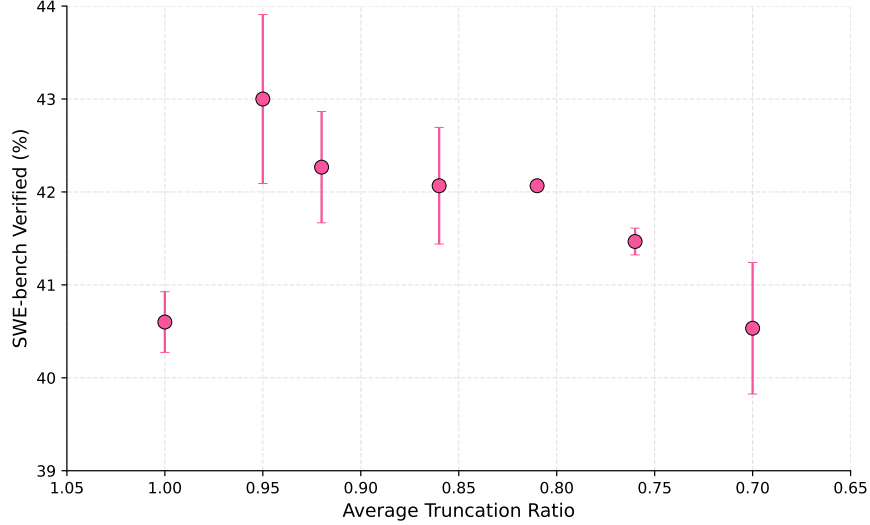
Figure 6 SWE-bench Verified performance vs. truncation ratio at 32K context. Each point represents 3,000 $\mathbf{T_1}$ trajectories partitioned by truncation ratio, averaged over 3 seeds. Trajectories with truncation ratio 0.95 perform best.

and leave further investigation for future work.

Fully Fit in 32K	Ordered Truncated	Random Truncated
$40.60\% \pm 0.69\%$	**43.00% $\pm$ 1.93%**	$37.47\% \pm 0.50\%$

Table 4 Effect of truncation strategy on SWE-bench Verified performance. All conditions use 3,000 $\mathbf{T_1}$ trajectories from GLM-4.5-Air, trained on Qwen 3-32B and evaluated at 32K context. Ordered truncation selects trajectories with the highest truncation ratios < 1, while random truncation samples arbitrarily from all trajectories exceeding 32K tokens. Results averaged over 3 seeds.

Assumptions were met. An independent samples t-test showed that ordered truncation ($M = 43.00$, $SD = 1.93$) significantly outperformed random truncation ($M = 37.47$, $SD = 0.50$), $t(4) = 4.81$, $p = .009$, $d = 3.93$. This confirms that the order in which content is truncated matters: preserving earlier turns in the trajectory is more effective than random truncation.

5.3 Data Filtering for Specialization

Long Edit and Tool Call Length Filtering: Table 5 highlights the effect of filtering out trajectories with long edits and large tool calls during specialization. We classify a excessively long edit as final edits exceeding 40 lines and excessively large tool calls as tool call responses containing more than 600 tokens. Ablating these filtering conditions for each repository, we find that no single filter setting reliably generalizes for all repositories. Indeed, filtering long edits works very well for Django and Sympy, but is ineffective for Sphinx, which instead benefits from filtering for tool call size.

We also apply these filtering techniques to $\mathbf{T_1}$ trajectories from **Sera-4.5A-Lite**, as shown in Table 6, which results in no significant improvements.

Taken together, our results suggest that filtering can be targeted to improve performance on specific repositories but is not as reliable in aggregate. The effectiveness of filtering methods likely reflects individual codebase characteristics. As a result, we suggest that users develop their own filtering heuristics for personal repositories.

Specializing to Multiple Repositories: We also train on Django and Sympy jointly to investigate whether SERA can be used to specialize to multiple codebases at once. We randomly sample half our initial Django and Sympy datasets from Section 4.3, training on the combined dataset. We jointly evaluate Django and Sympy instances separately, then average them with equal weighting, which debiases against the larger

Repository	No Filter	Patch ≤40 Lines	Both Filters
Django	$49.93\% \pm 1.64\%$	**$52.23\% \pm 1.64\%$**	$50.07\% \pm 2.95\%$
Sympy	$46.67\% \pm 2.67\%$	**$51.11\% \pm 1.54\%$**	$44.89\% \pm 1.54\%$
Sphinx	$32.29\% \pm 1.94\%$	$30.30\% \pm 6.95\%$	**$37.14\% \pm 6.95\%$**

Table 5 Effect of filtering on repository-specialized data. "Patch ≤40 Lines" drops trajectories with patches exceeding 40 lines. "Both Filters" additionally removes trajectories where average tool output exceeds 600 tokens. Filtering patches improves Django (+2.3%) and Sympy (+4.4%), while Sphinx benefits from the combined filter (+4.9%). Trained on Qwen 3-32B, evaluated at 32K context. Results averaged over 3 seeds.

No Filter	Patch ≤40 Lines	Tool Output ≤600
$43.93\% \pm 1.30\%$	$43.67\% \pm 1.60\%$	$44.00\% \pm 0.00\%$

Table 6 Effect of filtering on general T_1 trajectories from GLM-4.5-Air. "Patch ≤40 Lines" removes trajectories with patches exceeding 40 lines (n=5,364 from n=7,400). "Tool Output ≤600" removes trajectories where average tool output exceeds 600 tokens (n=6,136 from n=7,400). Neither filter improves performance on general data. Trained on Qwen 3-32B, evaluated at 32K context. Results averaged over 3 seeds.

number of Django instances in SWE-Bench Verified. We find that while performance drops slightly on each constituent codebase, the average performance of the combined dataset outperforms 10000 T_1 trajectories from **Sera-4.5A-Lite** (Table 7). In addition, independent performances on Django and Sympy still compare favorably to the teacher model. This indicates that **SERA** can be applied to multiple codebases for broadly improved performance, which reflects the needs of enterprises and larger research teams.

Training Data	Django	Sympy	Average
Specialized (8k Django)	**$52.23\% \pm 1.64\%$**	—	—
Specialized (8k Sympy)	—	**$51.11\% \pm 1.54\%$**	—
Specialized (4k Django + 4k Sympy)	$50.07\% \pm 0.50\%$	$46.22\% \pm 3.35\%$	**48.15%**
General (10k samples)	$45.60\% \pm 0.90\%$	$48.89\% \pm 0.77\%$	47.25%

Table 7 Multi-repository specialization results. Specialized training on 8,000 single-repository trajectories achieves the best per-repository performance. Mixed training (4,000 Django + 4,000 Sympy) achieves the best average. General training uses 10,000 T_1 trajectories from **Sera-4.5A-Lite**. Average is computed with equal weighting between Django and Sympy. Trained on Qwen 3-32B, evaluated at 32K context. Results averaged over 3 seeds.

5.4 Teacher Models

Early in Section 4.1, we show that GLM-4.5-Air is a much better teacher than Claude 3.7 Sonnet despite similar SWE-Bench Verified performance. We hypothesize that this is in part due to GLM-4.5-Air's reasoning traces, which are longer and significantly more elaborate.

To study the effect of reasoning traces, we train on 4200 T_2 trajectories from GLM-4.5-Air where we remove reasoning traces and leave only tool calls. In Table 8 we find that this significantly degrades performance compared to the unchanged trajectories. Our results confirm our hypothesis that high-quality reasoning traces are essential when distilling data for coding agents.

5.5 Rollout Mixing

In Section 4.3, we mix T_1 and T_2 trajectories to increase sample size during specialization. We repeat this experiment at a significantly larger scale in Table 9 using GLM-4.6 as a teacher. Combining 16,000 T_2 trajectories and 9,224 T_1 trajectories improves performance compared to training only on 16,000 T_2 trajectories. While the resulting model falls just shy of scaling only T_2 trajectories, the results suggest that T_1 and T_2 can be reliably mixed to extract further performance gains in data constrained settings.

Condition	SWE-bench Verified
With Reasoning	**41.00% ± 1.31%**
No Reasoning	23.00%

Table 8 Effect of reasoning traces on coding agent performance. Both conditions use 4,200 $\mathbf{T_2}$ trajectories from GLM-4.5-Air. No Reasoning removes all reasoning traces, retaining only tool calls. Trained on Qwen 3-32B, evaluated at 32K context. With Reasoning results averaged over 3 seeds.

	16k $\mathbf{T_2}$	**25k $\mathbf{T_2}$**	**16k $\mathbf{T_2}$ + 9k $\mathbf{T_1}$**
SWE-bench Verified	47.07% ± 1.21%	**49.53% ± 1.94%**	49.00% ± 1.64%

Table 9 Effect of mixing $\mathbf{T_1}$ and $\mathbf{T_2}$ trajectories. All data generated using GLM-4.6 as teacher. 16k $\mathbf{T_2}$ uses 16,000 second-rollout trajectories. 25k $\mathbf{T_2}$ uses 25,224 second-rollout trajectories. Mixed condition combines 16,000 $\mathbf{T_2}$ with 9,224 $\mathbf{T_1}$ trajectories. Mixing improves over 16k $\mathbf{T_2}$ alone (+1.9%) but falls slightly short of scaling to 25k $\mathbf{T_2}$. Trained on Qwen 3-32B, evaluated at 32K context. Results averaged over 3 seeds.

Assumptions of ANOVA are also met. A one-way ANOVA reveals no statistically significant difference between trajectory mixing strategies, $F(2,6) = 1.91$, $p = .229$, $\eta^2 = .39$. This indicates that $\mathbf{T_1}$ and $\mathbf{T_2}$ trajectories can be combined without degrading performance, enabling improved sampling efficiency.

6 Robustness of Evaluations

To assess the reliability of our findings, we conducted a systematic statistical analysis across all experiments in this paper. Our analysis aggregates within and between experiments and for multiple random seeds that include all experiments for scaling laws, verification thresholds, truncation strategies, specialization mixtures, filtering ablations, and baseline comparisons. In total, this analysis covers 78 experimental conditions, each evaluated with three random seeds, yielding 234 individual evaluation runs. Based on our findings we concluded with recommended best practices for coding agent evaluations.

Observed Variance: Across all experimental conditions, we observe standard deviations ranging from 0.5% to 3.0%, with a median of 1.2%. This is problematic when the magnitude of improvement in coding agent research is typically also 1–3%. Many reported gains in the literature fall within one standard deviation of run-to-run noise.

Signal-to-Noise Analysis: A practical way to assess whether an observed improvement is real is to compute the signal-to-noise ratio (SNR): the absolute difference between methods divided by the typical run-to-run variance. When SNR < 1, noise dominates and the result cannot be trusted. When SNR is between 1–2, the result is borderline and requires more seeds. When SNR > 2, there is likely a real effect. Applying this framework to our experiments:

- **High confidence (SNR > 3):** Specialized vs. general data (+4.3%, SNR=5.6), SERA vs. SWE-smith with same teacher (+4.7%, SNR=4.4), scaling law predictions (mean error 0.4%)
- **Moderate confidence (SNR 2--3):** Verification threshold equivalence (all within 2.9%, SNR confirms no difference), truncation ratio effects (+2.4%, SNR=2.2)
- **Low confidence (SNR < 2):** Student matching teacher at 8k samples (1.7% difference, SNR=1.4, error bars overlap)

How Many Seeds Do You Need? Based on the empirical variance in our data (median standard deviation of 1.2%), Table 10 shows approximately how many seeds are required to achieve SNR $\geq$ 2 for different effect sizes. These estimates follow directly from the definition: to achieve SNR = 2 for an effect of size δ, the standard error must be at most $\delta/2$, requiring $n \approx (2 \cdot \text{std}/\delta)^2$ seeds.

The Single-Seed Problem: Many published results in coding agent research report single-seed evaluations. Our data demonstrates the danger of this practice. Across multiple experiments, we find cases where different

Effect Size	Seeds for SNR ≥ 2	Reliability with $n = 3$
1%	~15	Cannot detect reliably
2%	~4	Borderline
3%	~2	Adequate
5%	~2	High confidence

Table 10 Seeds required to achieve SNR ≥ 2 for different effect sizes, derived from the empirical variance in our experiments (median std = 1.2%). With only 3 seeds, improvements below 2–3% should be treated with skepticism.

random seeds lead to opposite conclusions about which method is best. For example, in our truncation experiments, seeds 1 and 2 identify ratio 0.95 as optimal, while seed 3 identifies ratio 0.92 as optimal with 0.95 performing 2.2% worse. Single-seed ablations cannot be trusted.

Cross-Model Generalization: An concerning observation is that methods might not generalize well across different base models or teacher models. We observe that SWE-smith achieves 32.6% with Qwen 2.5-32B but only 25.3% with Qwen 3-32B. This 7.3% drop suggesting the method may have been unintentionally optimized for the earlier model family. For our method, changing the teacher model to Sonnet 3.7 and Sonnet 4.0 behaves as expected, demonstrating cross-model generalization. However, we did not have resources to test cross-model generalizations for the base model. Even our findings should be interpreted with caution: improvements we observe may not transfer to base models outside the Qwen and GLM families.

Scaling Laws as a Robustness Check: We found scaling laws to be invaluable for ensuring reliable results and recommend that future work in coding agents incorporate them where possible. Our scaling experiments (Figure 3) show that performance follows a highly predictable power law ($R^2 > 0.95$, mean prediction error 0.4%). Scaling laws provide several benefits: (1) *experimentation efficiency*: running experiments at smaller, cheaper scales while extrapolating findings to larger scales, since power laws have proven predictable and reliable; (2) *cost estimation*: predicting the resources required to reach target performance levels before committing to expensive runs; (3) *method comparison*: estimating sample efficiency and cost differences between methods without running exhaustive experiments at all scales; and (4) *robustness checking*: when a method's performance falls significantly outside the scaling law prediction, this signals either a genuine breakthrough or, more likely, noise or overfitting to a particular configuration.

Recommendations: Based on this analysis, we encourage researchers to (1) run a minimum of 3 seeds, preferably more for ablations expecting improvements below 3%, (2) report standard deviations alongside means, (3) compute the signal-to-noise ratio and treat SNR < 2 results as preliminary, (4) verify that improvements transfer across model configurations, and (5) fit scaling laws where feasible to enable efficient experimentation and robustness checking.

7 Deployment

As part of the Ai2 Open Coding Agents release, we provide a lightweight proxy server that enables Claude Code to use SERA as its backend. This section describes the key implementation considerations for deploying SERA in practice.

Tool Format Compatibility: SERA is trained on SWE-agent tool formats and performs best with exact format matching at inference time. Deploying the model with a different agent scaffold, or even subtle formatting differences, degrades performance significantly. Claude Code uses a different tool set (Read, Edit, Write, Bash) than SWE-agent (str_replace_editor, bash), so our sera-cli proxy translates between them. Path normalization is also required: data generation with SWE-agent uses a consistent working directory across all trajectories, so the proxy translates these paths to the user's current working directory. Tool result formatting must match training exactly including details like whitespace conventions and directory listing formats. When any of these conventions mismatch, the model can enter unproductive loops (e.g., repeatedly verifying edits that were already applied correctly), resulting in a poor experience for users. These issues are difficult to detect without agent scaffold specific evaluations.

Infrastructure: The proxy connects Claude Code to any OpenAI-compatible endpoint serving **SERA**. We use vLLM (Kwon et al., 2023) with the Hermes tool calling parser. For serverless deployment, we provide Modal integration scripts, though the model is portable to any cloud GPU provider or on-premises infrastructure. The proxy handles API translation (Anthropic format to OpenAI format), tool mapping, and response streaming and can be easily modified to handle **SERA** models specialized on a specific repository. **SERA** requires at least one 80GB GPU (e.g., A100 80GB, H100, or greater) for deployment. Quantization (e.g., AWQ, GPTQ) can further improve throughput and reduce memory requirements.

8   Related Work

Training data generation has emerged as a critical bottleneck for developing capable software engineering agents. Several approaches address this challenge through different methodologies for environment construction, data synthesis, and verification.

The most closely related work are other synthetic data generation approaches. such work includes, BugPilot (Sonwane et al., 2025a) which synthesizes bugs by instructing agents to add features, capturing unintentional test breakages as training data that more closely mirrors real development patterns. SWE-Synth (Pham et al., 2025) leverages LLM agents to simulate debugging workflows, producing bug-fix pairs with test cases and structured repair trajectories. SWE-Mirror (Wang et al., 2025) takes a different approach, distilling real issues from GitHub and mirroring them into repositories with configured environments, enabling data generation across multiple programming languages. SWE-smith (Yang et al., 2025) introduces an automated pipeline that synthesizes task instances by breaking existing tests in Python codebases, producing 50K instances from 128 repositories. R2E-Gym (Jain et al., 2025) introduces SYNGEN, a synthetic data curation recipe that uses test generation and commit back-translation to scale environment curation without relying on human-written issues. Skywork-SWE (Zeng et al., 2025) investigates data scaling laws for software engineering, demonstrating that model performance continues to improve with dataset size without saturation.

Environment construction has been addressed through multiple strategies: SWE-Gym (Pan et al., 2024) provides the first dedicated training environment with over 2,400 executable Python task instances for training language model-based software engineering agents. Repo2Run (Hu et al., 2025) uses LLM agents to iteratively build Docker environments from repository feedback, SWE-Factory (Guo et al., 2025) employs multi-agent collaboration with environment memory pools, and RepoST (Xie et al., 2025b) leverages sandbox testing to isolate functions and their dependencies for scalable construction.

Recent work has also explored optimizing the training process itself. SWE-Lego (Tao et al., 2026) demonstrates that a refined supervised fine-tuning procedure with error masking and difficulty-based curriculum can achieve state-of-the-art performance without reinforcement learning. SWE-Playground (Zhu et al., 2025) synthetically generates projects and tasks from scratch using strong language models, eliminating reliance on external data sources. Agent Data Protocol (Song et al., 2025) introduces a unified representation language for agent training datasets, enabling standardized fine-tuning across heterogeneous data sources from coding, browsing, and tool-use domains.

Evaluation Benchmarks: Standardized benchmarks have been essential for measuring progress in automated software engineering. SWE-bench (Jimenez et al., 2023) established the primary evaluation framework using real GitHub issues and pull requests from popular Python repositories. The benchmark has since been extended: SWE-rebench (Badertdinov et al., 2025) addresses contamination through continuous collection of fresh tasks, and Multi-SWE-bench (Zan et al., 2025) expands coverage to multiple programming languages including Java, TypeScript, Go, and Rust. Beyond repository-level tasks, BigCodeBench (Zhuo et al., 2024) evaluates models on tasks requiring diverse function calls from 139 libraries, while CodeRAG-Bench (Wang et al., 2024b) systematically studies how retrieval-augmented generation can improve code generation across basic programming to repository-level problems.

Agent Architectures: Various architectural approaches have been proposed for software engineering agents. SWE-agent (Yang et al., 2024) introduces a custom agent-computer interface designed for efficient repository navigation and code editing. OpenHands (Wang et al., 2024a) provides a modular platform supporting multiple agent implementations with standardized tool interfaces and sandboxed execution. Agentless (Xia et al., 2024) demonstrates that competitive performance can be achieved through a simpler three-phase

approach combining localization, repair, and validation, without complex agent scaffolding. OpenHands-Versa (Soni et al., 2025) further shows that a generalist agent with minimal tools—code editing, web search, and multimodal browsing—can achieve competitive performance across diverse benchmarks without domain-specific specialization.

Training Methods: Beyond data generation, several works explore training approaches for software engineering models. SWE-RL (Wei et al., 2025) applies reinforcement learning on open software evolution data using lightweight rule-based rewards to improve reasoning capabilities. SkyRL-Agent (Cao et al., 2025) provides efficient RL training infrastructure for multi-turn LLM agents. SWE-Fixer (Xie et al., 2025a) combines BM25-based file retrieval with a separate code editing module, training both components on 110K GitHub issues. These reinforcement learning approaches require substantial infrastructure for online rollouts and distributed training, which we discuss further in Section 2.

9 Limitations

There are several limitations of our work. While we draw certain best-faith conclusions based on our empirical results, this section is mostly about expression reasonable uncertainty and what this might mean for the interpretation of our results. We also highlight the gaps that might make our conclusions one-sided or biased due to the particular experiments that we ran.

Hard vs soft verification: We show that different verification levels, including no verification at all, perform approximately equally well. Conceptually, this is surprising. One explanation is that early performance gains on coding tasks depend primarily on learning skills like converting intentions into code edits and navigating codebases, rather than on code correctness. However, once a model saturates on these aspects, verified correct code may become necessary for further improvement. We could not test this hypothesis at our scale. It is possible that with larger models or more training data, soft verification no longer suffices and hard verification with correct code becomes essential.

Matching teacher performance: Our specialization results show that we can match or exceed teacher performance at around 8,000 samples per repository, and our scaling laws predict this trend continues with sufficient data. However, we could not verify whether this advantage scales further due to compute limitations. The practical takeaway may be that while exceeding teacher performance is possible, the gains are modest and likely level off. At that point, upgrading to a stronger teacher becomes more efficient than generating additional data.

Evaluation only on SWE-bench: We evaluate only on SWE-bench Verified. When using our model for our own coding tasks, we find it performs well but exhibits some undesirable behaviors leftover from training. For instance, it attempts to call a nonexistent submit tool when its finished editing in Claude Code. While this suggests our results may generalize to some degree, we have not validated our model on other coding benchmarks or tasks, and we do not know how well it performs more broadly.

Private repo specialization : We demonstrate specialization on Django, Sympy, and Sphinx because these repositories have test data that allows us to evaluate whether specialization works. However, these are public repositories likely included in base model training data. Our specialization experiments may therefore be biased. While specialization effects are well-studied in fine-tuning scaling laws and our results appear plausible, we have not verified specialization on truly private codebases that models have never seen because we have no evaluation data to test this directly.

Statistical robustness concerns of claims: As discussed in Section 6, many of our comparisons are underpowered with $n = 3$ seeds. Some reported effects may be noise rather than genuine improvements. We encourage readers to focus on large effects ($>3\%$) and treat smaller differences with appropriate skepticism.

Model-specific results: All experiments use Qwen-3 family of models as the base model and GLM-4.5-Air or GLM-4.6 as teachers. While we have some experiments with Claude 3.7 Sonnet and CLaude 4.0 Sonnet that hints at generalization of our method, we do not know whether our findings generalize to other model families when evaluated thoroughly. The concerns we raise about model-specific optimization in the evaluation section apply equally to our own work.

10 Broader Impact

We believe that the release of the Ai2 Open Coding Agents that include our SERA models will have significant impact on the research community by enabling research on coing agents without requiring large resources or complicated systems. We also believe private repo specialization will have a significant effect of how small organizations use coding agents. In this section, we extend these discussion and the effects our work has.

Democratizing Coding Agent Research: A central barrier to progress in coding agent research has been the prohibitive cost and infrastructure complexity required to train competitive models. The Ai2 Open Coding Agents initiative aims to remove these barriers, with **SERA** as its first release. Reinforcement learning approaches require teams of 12 or more researchers, clusters of 64+ GPUs, and months of engineering effort to build sandboxed execution environments. **SERA** reduces this to a team of five and 16 GPUs, with a total cost under $2,000. This shift makes coding agent research feasible for individual researchers, small academic labs, and institutions in regions without access to large-scale compute infrastructure. By releasing 200,000 trajectories, our training code, we aim to further lower the barrier to entry so that the study of coding agents is not concentrated among a handful of well-resourced industry labs.

Enabling Private Codebase Specialization: The ability to specialize a coding agent to a private codebase has significant implications for individuals and small companies. Currently, developers who want AI-assisted coding must send their proprietary code to cloud API providers, creating privacy and intellectual property concerns while these closed system are also not adapted to work well with private data. **SERA** enables a fundamentally different workflow: users can train a small, local model specialized to their own codebase without exposing their code to any third party. This is particularly relevant for startups with proprietary algorithms, regulated industries (healthcare, finance, defense) where code cannot leave secure environments, and open-source maintainers who want AI assistance tailored to their specific project conventions.

Open Science and Reproducibility: We release all components needed to reproduce and extend our work: training data, generation code, model weights, and evaluation scripts. Beyond enabling replication, this provides a shared foundation that other researchers can build upon without recreating expensive infrastructure from scratch. Our detailed cost analyses and scaling laws further serve the community by providing realistic expectations for resource planning and by identifying which experimental factors actually matter, potentially saving other groups from pursuing unproductive directions.

Acknowledgments

References

Axolotl maintainers and contributors. Axolotl: Open Source LLM Post-Training, 2023. URL https://github.com/axolotl-ai-cloud/axolotl.

I. Badertdinov, A. Golubev, M. Nekrashevich, A. Shevtsov, S. Karasik, A. Andriushchenko, M. Trofimova, D. Litvintseva, and B. Yangel. SWE-rebench: An Automated Pipeline for Task Collection and Decontaminated Evaluation of Software Engineering Agents. *ArXiv*, abs/2503.00000, 2025.

Z. Cai, M. Cao, H. Chen, K. Chen, et al. InternLM2 Technical Report. *ArXiv*, abs/2403.17297, 2024.

S. Cao, D. Li, F. Zhao, S. Yuan, S. Hegde, C. Chen, C. Ruan, T. Griggs, S. Liu, E. Tang, R. Liaw, P. Moritz, M. Zaharia, J. E. Gonzalez, and I. Stoica. SkyRL-Agent: Efficient RL Training for Multi-turn LLM Agent. *ArXiv*, abs/2511.16108, 2025.

A. Chandra, A. Agrawal, A. Hosseini, S. Fischmeister, R. Agarwal, N. Goyal, and A. Courville. Shape of thought: When distribution matters more than correctness in reasoning tasks, 2026. URL https://arxiv.org/abs/2512.22255.

J. Copet, Q. Carbonneaux, G. Cohen, J. Gehring, J. Kahn, J. Kossen, F. Kreuk, E. McMilin, M. Meyer, Y. Wei, D. Zhang, K. Zheng, et al. CWM: An Open-Weights LLM for Research on Code Generation with World Models. *ArXiv*, abs/2510.02387, 2025.

J. Da, C. J. Wang, X. Deng, Y. Ma, N. Barhate, and S. M. Hendryx. Agent-RLVR: Training Software Engineering Agents via Guidance and Environment Rewards. *ArXiv*, abs/2506.11425, 2025.

L. Engstrom, A. Ilyas, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry. Implementation Matters in Deep Policy Gradients: A Case Study on PPO and TRPO. *ArXiv*, abs/2005.12729, 2020.

T. Furlanello, Z. C. Lipton, M. Tschannen, L. Itti, and A. Anandkumar. Born Again Neural Networks. In *International Conference on Machine Learning ICML*, pages 1602–1611, 2018.

GLM-4.5 Team, A. Zeng, X. Lv, Q. Zheng, Z. Hou, B. Chen, et al. GLM-4.5: Agentic, Reasoning, and Coding (ARC) Foundation Models. *ArXiv*, abs/2508.06471, 2025.

L. Guo, Y. Wang, C. Li, P. Yang, J. Chen, W. Tao, Y. Zou, D. Tang, and Z. Zheng. SWE-Factory: Your Automated Factory for Issue Resolution Training Data and Evaluation Benchmarks. *ArXiv*, abs/2506.10954, 2025.

P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep Reinforcement Learning that Matters. In *Conference on Artificial Intelligence AAAI*, pages 3207–3214, 2018.

G. E. Hinton, O. Vinyals, and J. Dean. Distilling the Knowledge in a Neural Network. *ArXiv*, abs/1503.02531, 2015.

R. Hu, C. Peng, X. Wang, J. Xu, and C. Gao. Repo2Run: Automated Building Executable Environment for Code Repository at Scale. 2025.

N. Jain, J. Singh, M. Shetty, L. Zheng, K. Sen, and I. Stoica. R2E-Gym: Procedural Environments and Hybrid Verifiers for Scaling Open-Weights SWE Agents. *ArXiv*, abs/2504.07164, 2025.

C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *ArXiv*, abs/2310.06770, 2023.

R. Just, D. Jalali, and M. D. Ernst. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Symposium on Operating Systems Principles*, 2023.

M. Luo, N. Jain, J. Singh, S. Tan, A. Patel, et al. DeepSWE: Training a Fully Open-sourced, State-of-the-Art Coding Agent by Scaling RL. https://www.together.ai/blog/deepswe, 2025. Together AI Blog.

M. A. Merrill, A. G. Shaw, N. Carlini, B. Li, H. Raj, I. Bercovich, L. Shi, J. Y. Shin, T. Walshe, et al. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

J. Pan, X. Wang, G. Neubig, N. Jaitly, H. Ji, A. Suhr, and Y. Zhang. Training Software Engineering Agents and Verifiers with SWE-Gym. *ArXiv*, abs/2412.21139, 2024.

M. V. T. Pham, H. N. Phan, H. N. Phan, C. C. Le, T. N. Nguyen, and N. D. Q. Bui. SWE-Synth: Synthesizing Verifiable Bug-Fix Data to Enable Large Language Models in Resolving Real-World Bugs. *ArXiv*, abs/2504.14757, 2025.

A. Rastogi, A. Yang, A. Q. Jiang, A. H. Liu, A. Sablayrolles, et al. Devstral: Fine-tuning Language Models for Coding Agent Applications. *ArXiv*, abs/2509.25193, 2025.

Y. Song, K. Ramaneti, Z. Sheikh, Z. Chen, B. Gou, T. Xie, Y. Xu, D. Zhang, A. Gandhi, F. Yang, J. Liu, T. Ou, Z. Yuan, F. F. Xu, S. Zhou, X. Wang, X. Yue, T. Yu, H. Sun, Y. Su, and G. Neubig. Agent Data Protocol: Unifying Datasets for Diverse, Effective Fine-tuning of LLM Agents. *ArXiv*, abs/2510.24702, 2025.

A. B. Soni, B. Li, X. Wang, V. Chen, and G. Neubig. Coding Agents with Multimodal Browsing are Generalist Problem Solvers. *ArXiv*, abs/2506.03011, 2025.

A. Sonwane, I. White, H. Lee, M. Pereira, L. Caccia, M. Kim, Z. Shi, C. Singh, A. Sordoni, M.-A. Coté, and X. Yuan. BugPilot: Complex Bug Generation for Efficient Learning of SWE Skills. *ArXiv*, abs/2510.19898, 2025a.

A. Sonwane, I. White, H. Lee, M. Pereira, L. Caccia, M. Kim, Z. Shi, C. Singh, A. Sordoni, M.-A. Côté, and X. Yuan. Bugpilot: Complex bug generation for efficient learning of swe skills, 2025b. URL https://arxiv.org/abs/2510.19898.

C. Tao, J. Chen, Y. Jiang, K. Kou, S. Wang, R. Wang, X. Li, S. Yang, Y. Du, J. Dai, Z. Mao, X. Wang, L. Shang, and H. Bai. SWE-Lego: Pushing the Limits of Supervised Fine-tuning for Software Issue Resolving. 2026.

N.-A. Team, Y. Cai, L. Chen, Q. Chen, et al. Nex-N1: Agentic Models Trained via a Unified Ecosystem for Large-Scale Environment Construction. *ArXiv*, abs/2512.04987, 2025a.

Q. Team, A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, et al. Qwen2.5 Technical Report. *ArXiv*, abs/2412.15115, 2024.

Q. Team, A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, et al. Qwen3 Technical Report. *ArXiv*, abs/2505.09388, 2025b.

J. Wang, D. Zan, S. Xin, S. Liu, Y. Wu, and K. Shen. SWE-Mirror: Scaling Issue-Resolving Datasets by Mirroring Issues Across Repositories. *ArXiv*, abs/2509.08724, 2025.

X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. 2024a.

Z. Z. Wang, A. Asai, X. V. Yu, F. F. Xu, Y. Xie, G. Neubig, and D. Fried. CodeRAG-Bench: Can Retrieval Augment Code Generation? *ArXiv*, abs/2406.14497, 2024b.

Y. Wei, O. Duchenne, J. Copet, Q. Carbonneaux, L. Zhang, D. Fried, G. Synnaeve, R. Singh, and S. Wang. SWE-RL: Advancing LLM Reasoning via Reinforcement Learning on Open Software Evolution. *ArXiv*, abs/2502.18449, 2025.

R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh. BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies. In *ESEC/FSE*, 2020.

C. Xia, Y. Deng, S. Dunn, and L. Zhang. Agentless: Demystifying LLM-based Software Engineering Agents. *ArXiv*, abs/2407.01489, 2024.

C. Xie, B. Li, C. Gao, H. Du, W. Lam, D. Zou, and K. Chen. SWE-Fixer: Training Open-Source LLMs for Effective and Efficient GitHub Issue Resolution. *ArXiv*, abs/2501.05040, 2025a.

Y. Xie, A. Xie, D. Sheth, P. Liu, D. Fried, and C. P. Rosé. RepoST: Scalable Repository-Level Coding Environment Construction with Sandbox Testing. *ArXiv*, abs/2503.07358, 2025b.

J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. *ArXiv*, abs/2405.15793, 2024.

J. Yang, K. A. Lieret, C. E. Jimenez, A. Wettig, K. Khandpur, Y. Zhang, B. Hui, O. Press, L. Schmidt, and D. Yang. SWE-smith: Scaling Data for Software Engineering Agents. *ArXiv*, abs/2504.21798, 2025.

D. Zan, Z. Huang, W. Liu, H. Chen, L. Zhang, S. Xin, L. Chen, Q. Liu, X. Zhong, A. Li, S. Liu, Y. Xiao, L. Chen, Y. Zhang, J. Su, T. Liu, R. Long, K. Shen, and L. Xiang. Multi-SWE-bench: A Multilingual Benchmark for Issue Resolving. *ArXiv*, abs/2504.00000, 2025.

L. Zeng, Y. Li, Y. Xiao, C. Li, C. Liu, R. Yan, T. Wei, J. He, X. Song, Y. Liu, and Y. Zhou. Skywork-SWE: Unveiling Data Scaling Laws for Software Engineering in LLMs. *ArXiv*, abs/2506.19290, 2025.

Y. Zhu, A. Gandhi, and G. Neubig. Training Versatile Coding Agents in Synthetic Environments. *ArXiv*, abs/2507.00001, 2025.

T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul, S. Brunner, C. Gong, T. Hoang, A. Zebaze, X. Hong, W.-D. Li, J. Kaddour, M. Xu, Z. Zhang, P. Yadav, N. Jain, A. Gu, Z. Cheng, J. Liu, Q. Liu, Z. Wang, D. Lo, B. Hui, N. Muennighoff, D. Fried, X. Du, H. de Vries, and L. von Werra. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. *ArXiv*, abs/2406.15877, 2024.

A   Scaling Law and Data Points

We fit a power law to our cost-performance data to predict how SERA scales with additional investment. The scaling law takes the form:

$$y = c - a \cdot x^{-b}, \tag{9}$$

where y is the SWE-bench Verified resolve rate (%), x is the total training cost in thousands of dollars (including both data generation and training), c is the asymptotic performance ceiling as cost approaches infinity, a is a scaling coefficient controlling how far below the asymptote performance begins, and b is the power law exponent governing the rate of diminishing returns. The curve is fitted separately for each cost regime (vLLM self-hosting at \$0.187/sample and z.ai API at \$0.092/sample), yielding different (c, a, b) parameters since the same number of samples maps to different costs.

To predict the cost of matching a baseline system, we solve Equation 9 for x at the target performance level y^*:

$$x^* = \left(\frac{a}{c - y^*} \right)^{1/b}. \tag{10}$$

For example, Devstral-Small-2 achieves 50.0% and GLM-4.5-Air achieves 50.5% on SWE-bench Verified. Solving for these targets yields predicted costs of \$7K (z.ai API) or \$15K (vLLM) to match Devstral-Small-2, and \$9K (z.ai API) or \$19K (vLLM) to match GLM-4.5-Air. The fitted asymptote is approximately 70%, suggesting substantial headroom remains if data quantity is scaled further, though we note this extrapolation is uncertain as it extends well beyond our observed data range.

Table 11 provides the exact data points underlying the scaling law in Figure 3. All experiments use Qwen 3-32B as the base model trained on SERA data generated with GLM-4.5-Air as the teacher, evaluated on SWE-bench Verified at 32K context length. Each condition is evaluated over 3 random seeds. We report these values to enable other researchers to directly compare against our scaling curve without needing to read approximate values from the plot.

Samples	Seed 1	Seed 2	Seed 3	Mean (%)	Std (%)	Cost (vLLM)	Cost (z.ai)
400	34.40	33.00	33.00	33.47	0.81	\$75	\$37
750	36.80	35.00	37.40	36.40	1.25	\$140	\$69
1,500	38.20	40.20	38.20	38.87	1.15	\$280	\$138
3,000	40.60	37.80	40.60	39.67	1.62	\$560	\$275
4,200	40.60	45.80	39.00	41.80	3.56	\$784	\$386
7,400	43.20	45.40	43.40	44.00	1.22	\$1,382	\$679
16,000	47.00	47.00	45.80	46.60	0.69	\$2,987	\$1,469

Table 11 Exact scaling law data points for Figure 3. Performance is SWE-bench Verified resolve rate (%). Costs include both data generation and training. Per-sample cost is \$0.187 for vLLM self-hosting and \$0.092 for the z.ai API.

B   Additional Baseline Comparisons

Table 12 provides additional baseline comparisons that complement the main results in Table 2. We train Qwen 2.5-32B on SWE-smith, which performs much better than when transferred to Qwen 3. This suggests that SWE-smith is optimized for Qwen 2.5-32B. We also include the SERA result using GLM-4.5-Air as a teacher, which shows the substantial performance improvement from using a stronger teacher model compared to Claude 3.7.

C   Specialization Results at 64K Context

Table 13 presents specialization results evaluated at 64K context length. Because SERA models are trained at 32K context while competing models like Devstral-Small-2 are trained at 64K or longer contexts, SERA

Method	SWE-smith	SERA
Base model	Qwen 2.5-32B	Qwen 3-32B
Teacher	Claude 3.7	GLM-4.5-Air
Eval context size	32K	32K
Sample size	6402	4933
SWE-bench Verified	32.60%	38.47% ± 1.01%

Table 12 Additional baseline comparisons. SWE-smith with Qwen 2.5-32B shows the method was optimized for this model family. SERA with GLM-4.5-Air demonstrates the benefit of stronger teacher models.

underperforms at 64K evaluation despite matching or exceeding these baselines at 32K (Table 3). This context length mismatch explains the performance gap: our models have not learned to effectively utilize the additional context available at 64K tokens.

Model	Django (231)	Sympy (75)	Sphinx (44)
Qwen 3-32B-Django	56.56% ± 0.66%	-	-
Qwen 3-32B-Sympy	-	48.00% ± 4.62%	-
Qwen 3-32B-Sphinx	-	-	35.61% ± 1.31%
GLM-4.5-Air	58.58% ± 1.39%	56.00% ± 1.33%	48.87% ± 1.98%
Devstral-Small-2-24B	**62.63% ± 1.32%**	**56.24% ± 3.27%**	**53.79% ± 4.73%**

Table 13 Specialization results at 64K context. Fine-tuned SERA models underperform baselines at 64K because they are trained at 32K context, while Devstral-Small-2 is trained at longer contexts. Results averaged over three seeds.

D   Cost Breakdown

We assume a cost of $2 per H100 GPU-hour throughout this section, which reflects current cloud pricing for on-demand instances.

Reinforcement learning.   RL-based approaches for coding agents require substantial compute. SkyRL-Agent (Cao et al., 2025) reports 4,601 H100-hours to train SA-SWE-32B, yielding a cost of $9,202 and achieving 39.4% on SWE-bench Verified. For comparison, DeepSWE (Luo et al., 2025) requires 9,180 H100-hours ($18,360) to reach similar performance. SERA's total cost for data generation and training is 960 H100-hours ($1,920), making it 4.8× cheaper than SkyRL in raw compute and 9.6× cheaper than DeepSWE. However, SERA also achieves higher data efficiency. We fit a power law to SERA's cost-performance curve (Figure 3) and find that SERA reaches SkyRL's 39.4% at a cost of just $352 when self-hosting via vLLM, or $173 via the z.ai API. This yields a cost-to-performance efficiency of 26× (vLLM) or 53× (z.ai) compared to SkyRL.

Synthetic data generation.   Figure 3 shows scaling curves under three cost regimes: self-hosted inference via vLLM, and API-based inference using GLM-4.5-Air and GLM-4.6 through the z.ai API. To derive the API cost, we analyzed 100 randomly sampled trajectories from the SWE-smith trajectory dataset (Yang et al., 2025) to measure actual token consumption patterns. Each trajectory consists of multiple API calls where the conversation history grows with each turn. For a given API call, the model receives the full conversation history (cached input), the new tool result or observation (uncached input), and produces a response (output). We measured these components and rescaled to 32K context length to match our training setup, yielding an average of 35 API calls per trajectory.

Table 15 shows the per-trajectory cost breakdown across four configurations: SWE-smith using the Sonnet 3.7 API, SERA using the z.ai API with GLM-4.5-Air and GLM-4.6, and SERA self-hosted via vLLM. For the API-based methods, we show the token-level breakdown; for vLLM, we report the GPU cost directly.

Provider	Input (/MTok)	Cached (/MTok)	Output (/MTok)
Anthropic (Sonnet 3.7)	$3.00	$0.30	$15.00
z.ai (GLM-4.5-Air)	$0.20	$0.03	$1.10
z.ai (GLM-4.6)	$0.60	$0.11	$2.20
vLLM (self-hosted)	0.065 GPU-hours/trajectory × $2/GPU-hour		

Table 14 API pricing used for cost calculations.

Component	% Token Cost	SWE-smith (Sonnet 3.7)	SERA (GLM-4.5-Air)	SERA (GLM-4.6)	SERA (vLLM)
Cached input (context)	95.9%	$0.2247	$0.0225	$0.0824	—
New input (tool results)	3.1%	$0.0730	$0.0049	$0.0146	—
Output (generations)	1.0%	$0.1151	$0.0084	$0.0169	—
Issue creation	—	$0.0540	—	—	—
Inference subtotal	—	$0.4668	$0.0358	$0.1139	$0.1307
Training	—	$0.0560	$0.0560	$0.0560	$0.0560
Total per trajectory	—	**$0.5228**	**$0.0918**	**$0.1699**	**$0.1867**

Table 15 Cost breakdown per trajectory. Token cost percentages show the share of total billed tokens across 35 API calls per trajectory, rescaled to 32K context. See Table 14 for pricing details.

The dominant cost for API-based methods is the cached conversation context, which accumulates approximately 749K tokens across 35 API calls per trajectory. For Sonnet 3.7, even with prompt caching at $0.30/MTok, the cumulative context accounts for 54.4% of inference cost. Output tokens, though far fewer (7.7K per trajectory), are disproportionately expensive due to the higher output price ($15.00/MTok). SWE-smith additionally requires $0.054 per trajectory for synthetic issue creation. In total, SERA with GLM-4.5-Air via the z.ai API is 5.7× cheaper than SWE-smith with Sonnet 3.7, and 2.0× cheaper than self-hosting via vLLM. GLM-4.6 via the z.ai API costs $0.1699 per trajectory—3.1× cheaper than SWE-smith and 1.1× cheaper than vLLM self-hosting, while providing a stronger teacher model. The 3.2× higher inference cost of GLM-4.6 compared to GLM-4.5-Air ($0.1139 vs $0.0358) is partially offset by the fixed training cost ($0.056), yielding only a 1.85× increase in total per-trajectory cost. Importantly, these per-trajectory comparisons do not account for data quality. As shown in Table 2, SERA achieves higher performance per sample than competing methods, and GLM-4.6 produces higher-quality data than GLM-4.5-Air at comparable sample sizes. When we account for this by comparing the cost to reach equivalent performance levels using our scaling law, the effective advantages are substantially larger: SERA reaches SWE-smith's 32.6% (Qwen 2.5) performance at a cost of $60 (vLLM) or $29 (z.ai with GLM-4.5-Air), compared to SWE-smith's $3,395. This yields a cost-to-performance efficiency of 57× (vLLM) or 115× (z.ai with GLM-4.5-Air).

At scale, SERA requires approximately $1.5K to generate 16,000 trajectories via the z.ai API with GLM-4.5-Air, $2.7K with GLM-4.6, compared to $3.0K via vLLM and $8.4K via the Sonnet 3.7 API. The scaling law in Figure 3 predicts that with GLM-4.6 via the z.ai API, matching Devstral-Small-2 performance requires approximately $6K in data generation cost, compared to $23K with GLM-4.5-Air via the z.ai API and $47K with vLLM self-hosting.

However, we note important caveats for using commercial APIs in research. API pricing is subject to change, and providers may adjust model quality, rate limits, or availability without notice. This makes experiments difficult to reproduce exactly and can introduce confounding factors if model behavior shifts between experimental runs. For these reasons, APIs may not be suitable for rigorous scientific work that demands full reproducibility. We still encourage researchers to consider the vLLM backend with open-weight models, which provides complete control over the inference process and ensures consistent behavior across experimental runs. That said, for practitioners operating under cost constraints who need to generate training data quickly, a commercial API with cached input pricing offers a viable alternative at substantially reduced

cost.

E   Model Card: SERA-32B

This section provides the model card for SERA-32B, following standard practices for documenting machine learning models.

E.1   Model Overview

SERA-32B is the first model in Ai2's Open Coding Agents series. It is a state-of-the-art open-source coding agent that achieves 49.5% on SWE-bench Verified, matching the performance of much larger models like Devstral-Small-2 (24B) and GLM-4.5-Air (110B). SERA-32B was trained using Soft Verified Generation (SVG), a simple and efficient method that is $26\times$ cheaper than reinforcement learning and $57\times$ cheaper than previous synthetic data methods to reach equivalent performance. The total cost for data generation and training is approximately $2,000 (40 GPU-days).

- **Paper:** https://allenai.org/papers/opencodingagents
- **Code:** https://github.com/allenai/SERA
- **CLI:** https://github.com/allenai/sera-cli (https://pypi.org/project/ai2-sera-cli/)
- **Collection:** https://huggingface.co/collections/allenai/open-coding-agents

E.2   Model Variants

Model	HuggingFace	Base	Teacher	SWE-bench Verified
SERA-32B	allenai/SERA-32B	Qwen 3-32B	GLM-4.6	49.5% ± 1.9%
SERA-32B-GA	allenai/SERA-32B-GA	Qwen 3-32B	GLM-4.5-Air	46.6% ± 0.7%
SERA-8B	allenai/SERA-8B	Qwen 3-8B	GLM-4.6	31.7% ± 0.9%
SERA-8B-GA	allenai/SERA-8B-GA	Qwen 3-8B	GLM-4.5-Air	31.7% ± 0.4%

Table 16 SERA model variants. All results evaluated at 32K context length. Standard deviations computed over 3 random seeds.

E.3   Performance Comparison

Model	Type	SWE-bench Verified (32K)
SkyRL-8B	Open-source	9.4%
Nex-N1-8B	Open-source	20.3%
SERA-8B	**Open-source**	**31.7%**
Qwen 3-32B (base)	Open-weight	24.4%
SWE-smith	Open-source	32.6%
SkyRL-Agent	Open-source	39.4%
DeepSWE	Open-source	42.2%
SERA-32B	**Open-source**	**49.5%**
Devstral-Small-2 (24B)	Open-weight	50.0%
GLM-4.5-Air (110B)	Open-weight	50.5%

Table 17 Performance comparison on SWE-bench Verified at 32K context. Open-source: code, model weights, and data publicly available. Open-weight: model weights available but training data/code not fully released.

E.4   Quickstart

The easiest way to use SERA is with the `sera` CLI, which provides seamless integration with Claude Code:

```
# Install the CLI
uv tool install ai2-sera-cli

# Option 1: Deploy on Modal (recommended for trying out)
modal setup  # one-time setup
sera --modal

# Option 2: Use an existing endpoint
export SERA_API_KEY=<your_api_key>
sera --endpoint <endpoint_url>
```

The first run with `-modal` takes approximately 10 minutes to download the model (∼65GB) and compile. Subsequent runs start in 1–2 minutes.

E.5   Model Details

Property	Value
Developer	Allen Institute for AI (Ai2)
Authors	Ethan Shen, Daniel Tormoen, Saurabh Shah, Ali Farhadi, Tim Dettmers
Base Model	Qwen 3-32B
Teacher Model	GLM-4.6 (357B)
Model Type	Coding agent / Software engineering agent
Training Method	Supervised fine-tuning on synthetic agent trajectories
Context Length	32K tokens
License	Apache 2.0

Table 18 SERA-32B model details.

Hyperparameter	Value
Epochs	3
Learning Rate	1e-5
Weight Decay	0.01
Max Sequence Length	32,768 tokens
Training Framework	Axolotl
Inference Framework	vLLM
Compute	40 GPU-days (∼$2,000)

Table 19 SERA-32B training configuration.

E.6   Training Data

SERA-32B is trained on 200,000 synthetic coding agent trajectories generated using Soft Verified Generation (SVG). SVG is a two-rollout pipeline: (1) a teacher model makes a change to a codebase starting from a randomly selected function, (2) the trajectory is converted into a pull request description, (3) the teacher attempts to reproduce the change given only the PR description, and (4) patches are compared using line-level recall (no test execution required). This approach removes the need for test infrastructure and enables data generation from any repository. The training data was generated from 120 Python codebases using GLM-4.6 (357B) as the teacher model.

E.7 Intended Use

- **Automated software engineering:** Bug fixes, feature implementation, refactoring
- **Repository specialization:** Fine-tune on private codebases to create specialized coding agents ($\sim$8,000 trajectories / $1,300)
- **Research:** Studying coding agents, data generation methods, and agent behavior

E.8 Limitations

- **SWE-bench training artifact:** The model was trained on SWE-bench-style tasks and may attempt to call a nonexistent `submit` tool when finished editing. The sera-cli proxy handles this automatically.
- **Evaluation scope:** Only validated on SWE-bench Verified (Python repositories). Performance on other languages or benchmarks is unknown.
- **Teacher bound:** Performance is largely bounded by the teacher model (GLM-4.6) capability.
- **Statistical variance:** Results computed over 3 seeds. Effects smaller than 2–3% should be interpreted with caution.
- **Model-specific:** Experiments use Qwen 3 as the base model. Generalization to other model families is not validated.

E.9 Bias, Risks, and Limitations

Like any language model without safety filtering, SERA can be prompted to generate harmful or insecure code. Users should be aware of the following risks:

- **Code security:** May generate code with security vulnerabilities (e.g., injection attacks, insecure defaults). All generated code should be reviewed before deployment.
- **Accuracy:** May produce incorrect or buggy code. Outputs should be tested and verified.
- **Inherited biases:** May reflect biases present in the Qwen 3-32B base model and GLM-4.6 teacher model.
- **Misuse potential:** Could potentially be used to generate malicious code or identify vulnerabilities for exploitation.

E.10 Responsible Use

This model is intended for research and educational use. Users should adhere to Ai2's Responsible Use Guidelines (https://allenai.org/responsible-use). Key principles include: use the model for beneficial purposes, review and test all generated code before deployment, do not use to generate malicious software or exploit vulnerabilities, and consider the potential impact of automated code generation in your context.

E.11 Hardware Requirements

SERA-32B requires at least one 80GB GPU (A100 80GB, H100, or equivalent) for deployment at 32K context length. Quantization (AWQ, GPTQ) can reduce memory requirements if needed.

E.12 License

This model is licensed under Apache 2.0. It is intended for research and educational use and may be used commercially in accordance with Ai2's Responsible Use Guidelines.

E.13 Citation

```
@article{sera2026,
  title={SERA: Soft-Verified Efficient Repository Agents},
  author={Shen, Ethan and Tormoen, Daniel and Shah, Saurabh
        and Farhadi, Ali and Dettmers, Tim},
```

```
  year={2026},
  institution={Allen Institute for AI},
  url={https://allenai.org/papers/opencodingagents}
}
```

E.14   Contact

For questions or issues, please contact `ethans03@cs.washington.edu` or `dettmers@cmu.edu`, or open an issue at https://github.com/allenai/SERA/issues.