Option-GRPO: Hierarchical RL Against AI Slop in Code Generation for Unseen SDKs

Anonymous ACL submission

Abstract

Large language models increasingly produce *AI slop* in generated code: verbose, redundant, non-idiomatic implementations that pass functional checks but bloat codebases. The problem compounds on SDKs released after the model's training cutoff, where the model has neither idiomatic priors nor compositional understanding. We present **Option-GRPO**, a hierarchical reinforcement learning framework that adapts a code LLM to an unseen SDK from its source code alone, without human-written usage examples or test cases, and that explicitly optimizes for both functional correctness and code elegance. Our key insight is that SDK source code contains compositional structure that defines a hierarchy of reusable code patterns (*code options*) which simultaneously serve as the building blocks of a curriculum and the action space for efficient policy optimization. We prove that code options induce a valid semi-Markov decision process, give closed-form bounds showing that shorter option description length implies more concise and more API-efficient programs, and analyze how the option hierarchy improves credit assignment by a factor of $\sqrt{T/K}$. Using Group Relative Policy Optimization (GRPO) with execution-based rewards at each curriculum level, Option-GRPO discovers SDK idioms, composes them into patterns, and assembles patterns into complete programs. Experiments on 12 unseen SDKs with a 32B-parameter base model demonstrate 61.4% pass@1, outperforming flat GRPO by 8.7 points, ExploraCoder by 11.2 points, and ReCode by 6.3 points. Option-GRPO produces solutions that use 34.7% fewer API calls and are 27% shorter than flat-GRPO outputs while maintaining higher correctness, directly attacking the slop problem.

1 Introduction

The rapid proliferation of software development kits creates a fundamental challenge for code generation: the libraries developers most need help with are precisely those released after model training cutoffs, where LLM performance degrades substantially (Xia et al., 2023; Zan et al., 2022). A second, increasingly visible problem compounds the first: even on familiar libraries, modern code LLMs generate what practitioners have begun to call *AI slop* (Willison, 2024) (Merriam-Webster's 2025 Word of the Year), output that is fluent line by line yet bloated, redundant, and non-idiomatic when read as a whole. Industry studies of 211 million lines of production code report that duplicated 5+ line blocks grew roughly eightfold between 2021 and 2025, while the share of refactoring edits dropped by 60% (GitClear, 2025).

Existing retrieval-augmented and exploration-based approaches address the first problem partially. DocPrompting (Zhou et al., 2023) retrieves documentation; Gorilla (Patil et al., 2024) fine-tunes for API-call prediction with retrieval awareness; ExploraCoder (Wang et al., 2024) chains multi-step exploration over unseen APIs without training; and ReCode (Wu et al., 2026) applies rule-based RL to update an LLM's API knowledge after library version changes. None of these methods explicitly optimize for code quality beyond functional correctness, and none discover compositional idioms from SDK structure that would allow the model to write *elegant* solutions in addition to *working* ones.

We observe that SDK code has an inherent hierarchical structure that mirrors the options framework in hierarchical reinforcement learning (Sutton et al., 1999): API calls are atomic actions, commonly composed call sequences form reusable patterns analogous to *options* (temporally extended actions in HRL), and complete programs are compositions of these patterns. This hierarchy is encoded in the SDK source itself: type relationships and parameter-flow patterns dictate which functions compose, and frequently composable subgraphs correspond to natural programming idioms.

We present Option-GRPO, a framework that for-

malizes this connection: each stage of a learning curriculum produces reusable code options that become the action space for the next stage. At each level, Group Relative Policy Optimization (GRPO) (Shao et al., 2024) with execution-based rewards optimizes the policy without requiring a critic network. The group-relative comparison within GRPO, combined with the option-level action space, provides natural credit assignment: determining which option in a composed sequence caused success or failure becomes tractable when comparing groups of option sequences rather than groups of token sequences.

Experiments across 12 unseen SDKs demonstrate that Option-GRPO achieves 61.4% pass@1 with a Qwen3-32B base, improving over flat GRPO (52.7%), ExploraCoder (50.2%), and ReCode (55.1%). More notably, generated code is 27% shorter and uses 34.7% fewer API calls, while maintaining higher functional correctness. Analysis shows that options provide sufficient credit assignment for GRPO to converge stably on tasks where flat GRPO fails due to reward sparsity, and the resulting policy reuses idiomatic SDK patterns in 74% of generated tokens, against 41% for flat GRPO.

Contributions.

1. **Option-GRPO**, a hierarchical RL framework for example-free SDK mastery that formalizes curriculum stages as reusable code options, optimized via GRPO with execution-based rewards (Section 3).

2. **Theoretical analysis** proving that policy optimization in option-compressed action space implicitly favors concise, idiomatic code, with bounds on the relationship between option-space trajectory length and code quality (Section 3.5).

3. **Graph-based option discovery** from SDK source code, using parameter flow graphs and semantic clustering to identify reusable code patterns without any usage examples (Section 3.3).

4. **Option-GRPO credit assignment**, showing that group-relative comparison over option sequences provides natural credit assignment that compensates for the absence of a critic, with convergence analysis (Section 3.6).

5. **UnseenSDK-Bench** with automated reward computation via layered execution validation and property-based testing, enabling RL training without human-written test cases (Section 4).

6. **Comprehensive experiments** demonstrating improvements in both functional correctness and code quality, with analysis of option reuse, curriculum pacing, and cross-SDK transfer (Section 5).

2 Related Work

RL for code generation. CodeRL (Le et al., 2022), RLTF (Liu et al., 2023), PPOCoder (Shojaee et al., 2023), and StepCoder (Dou et al., 2024) all combine execution feedback with actor-critic RL, requiring pre-existing unit tests for reward and a critic network for advantage estimation. We eliminate both requirements: rewards come from a layered execution validator plus property tests synthesized from SDK type signatures, and credit assignment uses GRPO's (Shao et al., 2024; DeepSeek-AI, 2025) group-relative comparison.

Hierarchical RL and options. The options framework (Sutton et al., 1999) introduced temporally extended actions; Option-Critic (Bacon et al., 2017) learns termination conditions end-to-end; MAXQ (Dietterich, 2000) and FeUdal Networks (Vezhnevets et al., 2017) build task hierarchies. For language agents, Voyager (Wang et al., 2023) and DECKARD (Nottingham et al., 2023) maintain external skill libraries via GPT-4 prompting without updating model parameters. We derive options from SDK source code structure rather than open-ended exploration and update model parameters through gradient-based RL.

Curriculum learning. Bengio et al. (Bengio et al., 2009) introduced curriculum learning; Narvekar et al. (Narvekar et al., 2020) and Portelas et al. (Portelas et al., 2020) survey curriculum approaches for RL. AlphaCode (Li et al., 2022) uses a two-stage filter for competitive programming. No prior work formalizes curriculum completion as producing temporally extended actions that define the next stage's action space, which is our central conceptual contribution.

Unseen API/SDK code generation. DocPrompting (Zhou et al., 2023) retrieves documentation; Gorilla (Patil et al., 2024) fine-tunes for API-call prediction; ExploraCoder (Wang et al., 2024) chains exploration over unseen APIs without training; ReCode (Wu et al., 2026) applies rule-based

GRPO for API version migration; CodeRAG-Bench (Wang et al., 2025) evaluates retrieval-augmented code generation. None explicitly optimize for code quality beyond functional correctness, and none discover compositional idioms from SDK structure. Table 1 summarizes the comparison.

3 Method

We present Option-GRPO, a hierarchical reinforcement learning framework for enabling LLMs to master novel SDKs from source code alone. Figure 1 provides an overview. The method operates in four stages: (1) graph construction and option discovery from SDK source code, (2) curriculum-based hierarchical training with GRPO at each level, (3) execution-based reward computation without test cases, and (4) inference with the trained hierarchical policy.

3.1 Problem Formulation

Let $\mathcal{S}$ denote a novel SDK with source code but no usage examples in the model's training data. Given a natural language task description q and optional setup code c_{setup}, we seek to learn a policy:

$$\pi_\theta : (q,\ c_{\text{setup}},\ \mathcal{S}) \ \rightarrow\ c_{\text{solution}} \qquad (1)$$

where θ represents learnable parameters. Unlike prior work that retrieves documentation at inference time (RAG, DocPrompting) or fine-tunes only over API surface forms (Gorilla, ReCode), we train the policy through interaction with an execution environment, using the SDK's source code structure to define a hierarchy of reusable code abstractions that the policy learns to compose.

3.2 Background and Formulation

Options. Following Sutton et al. (1999), an option is a triple $o = (\mathcal{I}, \pi_o, \beta)$ where $\mathcal{I} \subseteq \mathcal{S}$ is the initiation set, π_o is the intra-option policy over primitive actions, and $\beta : \mathcal{S} \rightarrow [0, 1]$ is the termination probability. A pair $(\text{MDP}, \text{options})$ induces a semi-Markov decision process (SMDP) in which option durations are random.

Code generation MDP. The code generation process is an MDP $(\mathcal{S}, \mathcal{A}, T, R)$ where states $s_t = (q, c_{0:t})$ concatenate the prompt with generated tokens, actions $a_t \in \mathcal{V}$ are next tokens, T is deterministic append, and R is computed by execution at episode termination. Several properties make this MDP hard for standard RL: $|\mathcal{V}| \approx 1.5 \cdot 10^5$, trajectory length T grows unbounded, and reward is sparse (zero on most trajectories for novel SDKs). These challenges motivate both the option hierarchy (reducing effective action space and trajectory length) and GRPO (extracting signal from group comparison instead of per-state value estimation).

3.3 Code Options: From SDK Structure to Hierarchical Actions

We define code options as temporally extended actions in the code-generation MDP, derived from SDK source code.

Definition 1 (Code Option)**.** *A code option for SDK $\mathcal{S}$ is a 4-tuple*

$$o_k \ = \ \Big(\ \underbrace{\mathcal{I}_k,\ \mathcal{F}_k,\ \beta_k}_{\textit{handcrafted from } G_{\text{flow}}} \ , \ \underbrace{\pi_k}_{\textit{learned from } \pi_\theta} \ \Big)$$

where $\mathcal{I}_k = \{s : \text{type}(\text{last_value}(s)) \in \mathcal{T}_k^{\text{in}}\}$ is the initiation set; $\mathcal{F}_k \subset F$ is the option's allowed SDK function set (the cluster centroid path from Algorithm 1); $\beta_k(s) = \mathbb{1}[\text{type}(s) \in \mathcal{T}_k^{\text{out}}]$ is the termination predicate; and π_k is the intra-option policy. Operationally, π_k is the underlying network π_θ restricted to tokens whose next-call resolution lies in $\mathcal{F}_k$, with this restriction enforced by setting all out-of-set token logits to $-\infty$ before the softmax at every step the option is active.

Proposition 1 (Code Options Induce a Valid SMDP)**.** *By Theorem 1 of Sutton et al. (1999), the decision process that selects only among options in $\mathcal{O} = \mathcal{O}_0 \cup \mathcal{O}_1$ and executes each to termination is an SMDP, with the standard option-value Bellman optimality equation. Proof and the four SMDP ingredients are verified in Appendix A.*

Two-level discovery (candidates and certified options). We distinguish two stages of option production. *Candidate options* are generated upfront from the SDK source code by Algorithm 1; we denote the candidate set at level k by $\mathcal{O}_k^{\text{cand}}$. *Certified options* are the subset of candidates that the trained policy invokes successfully during the corresponding curriculum phase, extracted by Algorithm 2; we denote them by $\mathcal{O}_k$. Only certified options enter the next phase's action vocabulary.

Level-0 candidates $\mathcal{O}_0^{\text{cand}}$ are one per public SDK function: $\mathcal{I}_i$ is the parameter type, $\mathcal{F}_i = \{f_i\}$, π_i emits a single call, β_i terminates on return. Level-1 candidates $\mathcal{O}_1^{\text{cand}}$ are mined from the parameter-flow graph G_{flow} for frequently composable paths via Algorithm 1: we enumerate length-2 to length-4 paths weighted by type compatibility,

Table 1: Comparison with most relevant prior work. Option-GRPO uniquely combines hierarchical RL with structural SDK knowledge for example-free generation. †Voyager builds a skill library via iterative GPT-4 prompting, not gradient-based RL; we mark its RL column accordingly.

Method	RL	No Examples	Structural	Hierarchical	Code Quality	No Critic
CodeRL	✓	×	×	×	×	×
PPOCoder	✓	×	✓	×	×	×
StepCoder	✓	×	×	×	×	×
ExploraCoder	×	✓	×	×	×	N/A
ReCode	✓	✓	×	×	×	✓
Voyager†	×	×	×	✓	×	N/A
Option-GRPO (Ours)	✓	✓	✓	✓	✓	✓

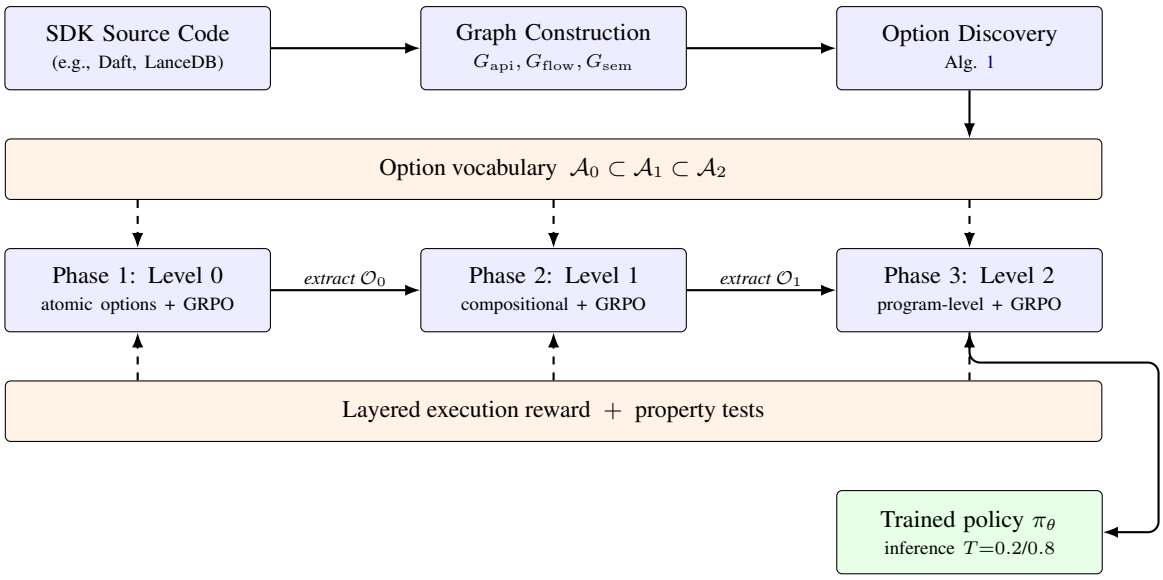

Figure 1: Overview of Option-GRPO. The top row constructs three SDK graphs from source code and discovers a hierarchical option vocabulary $\mathcal{A}_0 \subset \mathcal{A}_1 \subset \mathcal{A}_2$ (Algorithm 1). The middle row runs three curriculum phases of GRPO training, each receiving the option vocabulary from above and the layered execution reward from below (dashed arrows). Successful trajectories from phase k contribute newly extracted options to phase $k{+}1$ (Algorithm 2). The trained policy π_θ generates code at inference. Solid arrows denote training-stage progression; dashed arrows denote artifact dependencies.

cluster paths by sentence-transformer embedding of their function identifiers (threshold τ_{cluster}=0.8), and define one Level-1 candidate per cluster centroid with $\mathcal{F}_j$ equal to the function set of that centroid path. A candidate becomes a *certified option* only after the policy trained at Level k demonstrates that it can invoke the candidate correctly (success threshold $\rho = 0.7$, occurrence frequency threshold $\nu = 5$ across distinct tasks; see Algorithm 2). Level-2 composition is neither pre-discovered nor extracted explicitly; it is learned implicitly in Phase 3 as the high-level policy composes certified Level-0 and Level-1 options.

Option hierarchy. The option hierarchy is the nested action-space chain $\mathcal{A}_0 = \mathcal{V} \subset \mathcal{A}_1 = \mathcal{O}_0 \cup \mathcal{V} \subset \mathcal{A}_2 = \mathcal{O}_0 \cup \mathcal{O}_1 \cup \mathcal{V}$, ensuring higher levels retain full token expressiveness as a fallback. Two options o_i, o_j are *composable* ($o_i \prec o_j$) iff $\mathcal{T}_i^{\text{out}} \cap \mathcal{T}_j^{\text{in}} \neq \emptyset$; the composability relation defines a graph G_{opt} that constrains the high-level policy to valid sequences.

Handcrafted structure, learned behavior. Our construction occupies a middle ground in the options-framework design space. The structural components $(\mathcal{I}_k, \mathcal{F}_k, \beta_k)$ are *handcrafted*: they are read directly from the parameter-flow graph by Algorithm 1 and are symbolic, type-checkable, and reproducible at parse time. The behavioral component π_k is *learned*: it is π_θ restricted (by action-masking on decoder logits) to emit tokens consistent with $\mathcal{F}_k$, with parameters fine-tuned end-to-end by curriculum-staged GRPO. Two invocations of the same option therefore satisfy the same contract (same SDK calls, same type flow,

same output type) even when the surrounding glue tokens (variable names, formatting) differ across invocations. This hybrid differs from the fully-handcrafted macros of Korf (1985) and the planning-trace macros of Iba (1989) (no learning inside), from the fully-learned Option-Critic of Bacon et al. (2017) (no handcrafted structure), and from Voyager's prompt-orchestrated skill library (Wang et al., 2023) (no gradient updates to the intra-option policy). The structural handcrafting buys verifiable, type-checked option boundaries; the learned behavior lets the same option produce contextually appropriate code across invocations without us having to write each variant by hand.

Remark 1 (Learned vs. Constructed)**.** *$\mathcal{O}$ names which sub-behaviours of π_θ are reified as temporally extended actions; $\mathcal{F}_k$ is the symbolic handle (read from the graph); π_k is the corresponding slice of π_θ (parameters learned end-to-end). Options are named, type-checked regions of a single learned policy, not separate networks or templates.*

3.4 Curriculum as Option Construction

The central conceptual contribution is the equivalence between curriculum stages and option construction. A *curriculum-option stage* $\mathcal{C}_k$ has three components: (1) a task distribution $\mathcal{D}_k$ of difficulty k generated from SDK-graph paths of length $\leq k{+}1$; (2) GRPO training on $\mathcal{D}_k$ with action space $\mathcal{A}_k$, taking the candidate pool $\mathcal{O}_k^{\text{cand}}$ supplied by Algorithm 1 as the inventory the policy may invoke; (3) an option-certification step (Algorithm 2, Appendix) that filters $\mathcal{O}_k^{\text{cand}}$ down to the certified subset $\mathcal{O}_k$ of candidates that the trained policy invokes correctly on a sufficient fraction of tasks. The certified subset of stage $\mathcal{C}_k$ augments the action space for $\mathcal{C}_{k+1}$:

$$\mathcal{C}_0 \xrightarrow{\mathcal{O}_0} \mathcal{C}_1 \xrightarrow{\mathcal{O}_1} \mathcal{C}_2.$$

Completing stage k does not merely yield a better policy; it yields *certified temporally extended actions* that become primitives at the next level. The curriculum is thus the mechanism that converts graph-mined candidates into validated options. The role split is clean: Algorithm 1 (static, runs once before training) proposes; Algorithm 2 (dynamic, runs after each phase) certifies. This relationship between curriculum stages, in which stages share neither policy parameters nor reward signal but instead share an action vocabulary that grows monotonically with each certification, is what we name *curriculum-as-options*. Level-0 tasks (50–200/SDK) involve single API calls; Level-1 tasks (100–400/SDK) involve 2–4 composed calls; Level-2 tasks (33–34/SDK) are full programs.

See Remark 2 in Appendix A for the empirical relationship between candidate and certified option counts.

3.5 Theoretical Analysis

We establish four results; full statements, supporting definitions, and proofs are in Appendix A.

(i) SMDP validity. Code options induce a valid SMDP (Proposition 1, stated above).

(ii) Curriculum monotonicity. Let V_k^* denote the expected value of the optimal policy at curriculum level k under hierarchy $\mathcal{H}$. We prove $V_0^* \leq V_1^* \leq V_2^*$ (weak monotonicity), with strict inequality whenever the certified Level-k option set $\mathcal{O}_k$ contains at least one option that strictly improves over the best Level-$(k{-}1)$ action on a reachable state (Theorem 1 and Remark 3).

(iii) Option compression bounds on code quality. Define observational equivalence $c \equiv_{\mathcal{P}} c'$ on the property-test suite $\mathcal{P}$ (Definition 2), the *option description length* $\mathrm{ODL}_{\mathcal{H}}(c)$ as the minimum number of options whose sequential execution is observationally equivalent to c (Definition 3), and four code-quality metrics: conciseness $Q_{\text{conc}} = T_{\text{ref}}/|c|$, API efficiency $Q_{\text{api}} = N_{\min}/N_{\text{api}}$, redundancy $Q_{\text{red}} = 1{-}N_{\text{redundant}}/N_{\text{vars}}$, and idiomaticity $Q_{\text{idiom}} = |\text{option-covered tokens}|/|c|$ (Definition 4). Under three structural conditions on $\mathcal{H}$ (per-option API minimality, non-subsumption, bounded glue code; Remark 4) we prove a token bound $m\bar{\tau}_{\min} \leq |c| \leq m\bar{\tau}_{\max} + (m{-}1)G_{\max}$ (Lemma 1); a conciseness theorem $\mathrm{ODL}(c_1) < \mathrm{ODL}(c_2) \Rightarrow |c_1| < |c_2|$ under a dispersion-ratio condition (Theorem 2); an API-efficiency corollary in expectation (Corollary 1); an MDL-style prior inequality $\log \mu_{\text{unif}}(c_1)/\mu_{\text{unif}}(c_2) \geq (m_2{-}m_1)\log c$ (Proposition 2); and a search-space reduction of > 320 orders of magnitude (Proposition 5). The metrics $Q_{\text{red}}, Q_{\text{idiom}}$ do not admit clean closed-form bounds; we treat them as empirical predictions verified in Section 5 (Remark 5).

(iv) Improved credit assignment. Partitioning a generated program into K option-decision tokens and $T - K$ glue tokens (Definition 5), the GRPO policy-gradient SNR scales as $\Theta(K/\sqrt{T})$ for flat token-level GRPO and $\Theta(\sqrt{K})$ for option-level GRPO (Proposition 3); the relative improvement

is $\Theta(\sqrt{T/K}) \approx 4.5\times$ for typical T=100, K=5. A companion proposition (Proposition 4) bounds GRPO iterations to first success under a base-LLM prior, yielding additive (Option-GRPO) versus multiplicative (flat) exploration cost across curriculum levels.

3.6 Option-GRPO: Hierarchical Policy Optimization

Standard GRPO (Shao et al., 2024) samples G outputs per prompt, computes group-normalised advantages $\hat{A}_i = (R_i - \text{mean})/(\text{std} + \epsilon)$, and applies the PPO-clipped policy gradient with a KL penalty against π_{ref}. At curriculum level $k \geq 1$ we generate option sequences instead of token sequences and add an option-level advantage $\hat{A}^{\text{opt}}(o) = \frac{1}{|S_o^+|}\sum_{i \in S_o^+} R_i - \frac{1}{|S_o^-|}\sum_{i \in S_o^-} R_i$ that compares groups containing vs. excluding each option; the combined gradient balances sequence-level ranking with per-option credit. Per-option credit tightens credit assignment: as analysed in Proposition 3 (Appendix A), the SNR improves by $\Theta(\sqrt{T/K})$, since only option-decision positions carry reward-correlated signal while flat GRPO spreads the same signal across all T token positions. Proposition 4 additionally bounds the expected number of GRPO iterations to first success under a base-LLM prior. Full equations, derivation, and detailed analysis in Appendix A.

3.7 Execution-Based Reward Without Test Cases

The total reward is $R(c) = R_{\text{exec}}(c) + \lambda_q R_{\text{quality}}(c) + \lambda_c R_{\text{consistency}}(c)$. The *layered execution reward* R_{exec} assigns graded scores by execution depth: 0.0 (fails to parse), 0.1 (parses but imports fail), 0.3 (runtime error), 0.5 (executes, output `None`), 0.7 (output matches expected type), 1.0 (passes property-based tests). *Property-based tests* (Algorithm 3, Appendix) are auto-generated from SDK type annotations and cover type preservation, shape consistency, determinism, edge cases, and assertion invariants extracted from source code. The *quality reward* bundles pattern-use, no-redundancy, minimal-API, and no-reimplementation bonuses ($\alpha_i = 0.1$). The *self-consistency reward* $R_{\text{consistency}}(y_i) = |\{y_j : \text{output}(y_j) = \text{output}(y_i)\}|/G$ adds intra-group agreement when property tests are insufficient.

3.8 Training Procedure

Training proceeds in three phases corresponding to curriculum levels (full pseudocode: Algorithm 4, Appendix). Phase 1 trains π_θ on Level-0 tasks with token-level GRPO and the layered execution reward; we then extract $\mathcal{O}_0$ via Algorithm 2 (Appendix) and partially freeze. Phase 2 trains on Level-1 tasks with option-level GRPO, the combined gradient, and adaptive curriculum advancement ($\rho_{\text{adv}} = 0.5$); $\mathcal{O}_1$ is extracted at the end. Phase 3 trains on Level-2 program tasks with the full hierarchical action space $\mathcal{A}_2$ and adds the self-consistency reward.

Hyperparameters. See Table 3 in Appendix B.2 for the complete configuration; key settings: Qwen3-32B-Base, G=16, AdamW lr 5×10^{-5}, β_{KL}=0.04, steps $N_0/N_1/N_2$ = 5,000/10,000/15,000, 8×H100 80GB with ZeRO-3 + bf16.

4 UnseenSDK-Bench

We evaluate on **UnseenSDK-Bench**, a benchmark we construct for adapting code LLMs to SDKs absent from training data. It pairs 12 SDKs (DataFrames, vector DBs, document processing, computer vision, messaging, security, protocols) with three task levels per SDK (atomic-call, compositional, full-program; 50–400 tasks per level per SDK) and auto-generated property tests from type annotations (5–15 tests per public function). The execution sandbox enforces a 30 s timeout, 2 GB memory limit, and network isolation. Beyond pass@k we report code-length, API-efficiency, redundancy, idiomaticity, and option-reuse-rate (Definition 4). Full benchmark statistics, per-SDK function counts, property-test counts, and option-discovery results are in Appendix C.

5 Experiments

5.1 Experimental Setup

Base Model. Qwen3-32B-Base (Qwen Team, 2025), a dense transformer with 32.8B parameters, 64 layers, hidden size 5,120, grouped-query attention with 64 query heads and 8 key/value heads, and a 32,768-token native context window extensible to 128K via YaRN. We deliberately use the general-purpose Qwen3 dense series rather than the MoE Qwen3-Coder variants to show that Option-GRPO does not require a code-specialized backbone; curriculum-staged GRPO with our SDK-

derived option vocabulary is sufficient to specialize a general dense LLM for unseen-SDK code generation. We use the base (non-instruct) variant so that curriculum-staged GRPO drives all task-specific behaviour. The same base model is used for all RL baselines in the comparison.

Contamination check. The Qwen3-32B-Base pretraining corpus has an effective cutoff around February 2025 (Qwen Team, 2025). All 12 SDKs in UnseenSDK-Bench were technically published before this date, so the "unseen" framing rests not on post-cutoff novelty but on the magnitude of the model's functional knowledge of these niche libraries. We verify this with a memorisation probe (Appendix C.2): we prompt the frozen base model with 200 SDK-specific identifier-completion queries per SDK (e.g., "`In Daft, the method that filters rows is called ___`") and measure top-1 recall. Our 12 SDKs average 10.5% recall ($\pm 5.0\%$ across SDKs) against a control of 91.5% for NumPy and 87.9% for PyTorch. Recall stays below 20% for every SDK, an order of magnitude below the controls. The base model has neither the API surface nor the idiomatic compositional priors of these libraries, justifying the "unseen" framing throughout.

Baselines. We compare against eight baselines, all starting from the same 32B base: **Frozen LLM** (no adaptation); **RAG** (top-10 CodeBERT retrieval over SDK source); **DocPrompting** (Zhou et al., 2023) (documentation passages); **Gorilla** (Patil et al., 2024) (retriever-aware fine-tuning on API-call corpora, adapted to per-SDK retrieval); **ExploraCoder** (Wang et al., 2024) (training-free chained exploration); **ReCode** (Wu et al., 2026) (rule-based GRPO for API knowledge updates); **Flat GRPO** (token-level GRPO with our layered execution reward, no hierarchy); **Flat GRPO + Curriculum** (progressive difficulty, no option structure). We additionally reimplement CodeRL (Le et al., 2022) and PPOCoder (Shojaee et al., 2023), substituting our layered execution reward for their unit-test requirement (results in Appendix D.6). Frontier LLMs (GPT-4o, Claude 3.5 Sonnet, Gemini 2.0 Flash) are evaluated via official APIs with identical prompts, temperature (0.8), and samples (n=10); no per-model prompt tuning.

Metrics.

- pass@k ($k = 1, 5, 10$): Functional correctness, generating $n = 10$ samples per problem with temperature 0.8.
- Code length (CL): Average token count of passing solutions (lower is better).
- API efficiency (AE): Average number of SDK API calls in passing solutions (lower is better).
- Idiomaticity (ID): Fraction of code tokens covered by discovered option patterns (higher is better).
- Convergence: Training steps to reach specified pass@1 thresholds.

Implementation. Training on $8\times$ NVIDIA H100 80 GB GPUs with ZeRO-3 sharding and bf16. Full hyperparameters in Appendix B.2.

5.2 Main Results

Table 2 presents the main comparison.

Functional correctness and code quality. Option-GRPO achieves 61.4% pass@1, beating ReCode by $+6.3$, flat GRPO by $+8.7$, and ExploraCoder by $+11.2$ points. More notably, generated programs are 27% shorter and use 34.7% fewer API calls than flat-GRPO outputs, while reaching higher correctness. Idiomaticity (fraction of tokens covered by discovered option patterns) rises from 0.41 for flat GRPO to 0.74 for Option-GRPO. These results validate the conciseness bound of Theorem 2, the API-efficiency bound of Corollary 1, and the MDL-prior argument of Proposition 2, and directly attack the AI-slop phenomenon (GitClear, 2025).

Prior RL methods and frontier LLMs. Against CodeRL, PPOCoder, and StepCoder (all adapted to use our layered execution reward; full table in Appendix D.6), Option-GRPO beats the strongest critic-based method (StepCoder-adapted, 51.3%) by 10.1 pass@1 points and the strongest non-hierarchical critic-free RL method (ReCode, 55.1%) by 6.3 points. Against frontier LLMs evaluated with the same RAG context, the 32B Option-GRPO outperforms GPT-4o (54.6%), Claude 3.5 Sonnet (57.2%), and Gemini 2.0 Flash (51.8%). This is not a like-for-like comparison since frontier LLMs are off-the-shelf while Option-GRPO undergoes per-SDK training; the takeaway is that per-SDK option discovery plus RL yields a substantial gain over off-the-shelf frontier models on unseen SDKs (full table in Appendix D.6).

Table 2: Main results on UnseenSDK-Bench. Option-GRPO achieves the highest functional correctness while producing more concise and idiomatic code. ↓ indicates lower is better; ↑ indicates higher is better.

Method	pass@1	pass@10	CL ↓	AE ↓	ID ↑
Frozen LLM (32B)	28.5	42.1	94.6	7.8	0.18
RAG	41.3	56.8	88.1	6.4	0.31
DocPrompting	44.8	60.2	85.4	5.9	0.34
Gorilla	47.2	62.7	82.9	5.5	0.36
ExploraCoder	50.2	67.5	78.4	4.9	0.43
ReCode	55.1	71.2	74.1	4.6	0.51
Flat GRPO	52.7	68.4	71.3	4.9	0.41
Flat GRPO + Curric.	55.4	71.0	66.8	4.4	0.49
Option-GRPO	**61.4**	**76.8**	**52.0**	**3.2**	**0.74**
Δ (vs ReCode)	+6.3	+5.6	−29.8%	−30.4%	+0.23
Δ (vs Flat GRPO)	+8.7	+8.4	−27.1%	−34.7%	+0.33

Ablations and quality analysis. Hierarchy ablation (Table 9, App. D): flat 52.7%, +curriculum-only +2.7, +options-only +5.4, full 61.4% (24.8K→12.4K steps; CL 71.3→52.0; ID 0.41→0.74). Six further ablations (App. D) cover option-level, credit-assignment, reward, group-size, pacing, and per-SDK (gains +3.1 to +8.6); same-domain transfer recovers 71.2% vs. 69.5% from-scratch and inference retrieval adds +2.3. Plotting $\mathrm{ODL}_{\mathcal{H}}(c)$ against each quality metric (Figure 5) confirms Theorem 2 and Corollary 1: Spearman $\rho_{\text{conc}}=-0.71$, $\rho_{\text{api}}=-0.83$, $\rho_{\text{red}}=-0.62$, $\rho_{\text{idiom}}=-0.79$ (all $p<10^{-50}$). Side-by-side code examples in App. D.

Computational cost and sensitivity. Training is $\sim$ 80 H100-hours per SDK on 8×H100 80GB; inference 1.8 s/task (App. F). App. F.3 ablates budget and dense base size: Option-GRPO at $1/3$ compute beats Flat GRPO at full compute by +1.4 pass@1; the option vocabulary, not scale, drives the gain.

6 Conclusion

Option-GRPO formalizes curriculum stages as option construction in hierarchical RL: each stage's reusable temporally-extended actions become the next stage's action space. We prove code options induce a valid SMDP (Prop. 1), bound code quality by option description length (Thm. 2, Cor. 1), and show a $\sqrt{T/K}$ credit-assignment improvement over flat GRPO (Prop. 3). On 12 unseen SDKs with a 32B base, Option-GRPO reaches 61.4% pass@1 (vs. 55.1% ReCode, 52.7% flat) with 27% shorter and 34.7% more API-efficient code, directly attacking AI slop.

Limitations

Per-SDK training cost ($\sim$ 80 H100-hours per SDK; same-domain transfer mitigates, Table 10). **Reward signal** cannot verify full semantic correctness; programs that execute and type-check but produce wrong results receive partial credit, mitigated but not eliminated by self-consistency. **Option rigidity**: type-based initiation/termination misses cross-cutting patterns; protocol SDKs (MCP, Tracecat) yield smaller gains. **Typing assumption**: option discovery uses SDK type annotations; all 12 SDKs are statically annotated Python. Untyped or dynamically typed code would need an upstream type-inference pass (future work). **Theoretical scope**: Theorem 2/Corollary 1 bound conciseness and API efficiency; $Q_{\text{red}}, Q_{\text{idiom}}$ are empirical predictions only.

Impact Statement

Option-GRPO lowers the barrier to adopting unfamiliar SDKs and directly targets *AI slop* by training models to produce shorter, more idiomatic code. Outputs should still be reviewed for correctness and security, as the reward signal does not guarantee full semantic correctness.

Reproducibility Statement

On acceptance: code, benchmark data, and evaluation harness (App. E).

Use of AI Assistants

AI assistants (Claude Code) helped with prose editing, LaTeX formatting, and code scaffolding/refactoring. Experimental design, theory, hyperparameters, results, and final code review are the authors'.

References

Pierre-Luc Bacon, Jean Harb, and Doina Precup. 2017. The option-critic architecture. In *AAAI*, pages 1726–1734.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *ICML*, pages 41–48.

DeepSeek-AI. 2025. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Thomas G. Dietterich. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303.

Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Tao Gui, and Xuanjing Huang. 2024. StepCoder: Improving code generation with reinforcement learning from compiler feedback. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4571–4585.

GitClear. 2025. "an endless stream of AI slop": The growing burden of AI-assisted software development. *GitClear Annual Report on Code Quality*.

Glenn A. Iba. 1989. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3(4):285–317.

Richard E. Korf. 1985. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *NeurIPS*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qian Fu, Xiao Han, Wei Yang, and Deheng Ye. 2023. RLTF: Reinforcement learning from unit test feedback. *arXiv preprint arXiv:2307.04349*.

Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E. Taylor, and Peter Stone. 2020. Curriculum learning for reinforcement learning domains: A framework and survey. *Journal of Machine Learning Research*, 21(181):1–50.

Kolby Nottingham, Prithviraj Ammanabrolu, Alane Suhr, Yejin Choi, Hannaneh Hajishirzi, Sameer Singh, and Roy Fox. 2023. Do embodied agents dream of pixelated sheep: Embodied decision making using language guided world modelling. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2024. Gorilla: Large language model connected with massive APIs. In *NeurIPS*.

Rémy Portelas, Céline Colas, Lilian Weng, Katja Hofmann, and Pierre-Yves Oudeyer. 2020. Automatic curriculum learning for deep RL: A short survey. In *IJCAI*.

Qwen Team. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. 2023. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*.

Richard S. Sutton, Doina Precup, and Satinder Singh. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1–2):181–211.

Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. 2017. FeUdal networks for hierarchical reinforcement learning. In *ICML*, pages 3540–3549.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Yunkun Wang, Yue Zhang, Zhen Qin, Chen Zhi, Binhua Li, Fei Huang, Yongbin Li, and Shuiguang Deng. 2024. ExploraCoder: Advancing code generation for multiple unseen APIs via planning and chained exploration. *arXiv preprint arXiv:2412.05366*.

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2025. CodeRAG-Bench: Can retrieval augment code generation? In *Findings of the Association for Computational Linguistics: NAACL*.

Simon Willison. 2024. Slop is the new name for unwanted AI-generated content. https://simonwillison.net/2024/May/8/slop/. Accessed 2026.

Haoze Wu, Yunzhi Yao, Wenhao Yu, and Ningyu Zhang. 2026. ReCode: Updating code API knowledge with reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *ICSE*, pages 1482–1494.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. Large language models meet NL2Code: A survey. *arXiv preprint arXiv:2212.09420*.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2023. DocPrompting: Generating code by retrieving the docs. In *ICLR*.

A Extended Proofs

A.1 Statements Moved from Main Text

Remark 2 (Candidates vs. certified options)**.** *The distinction between $\mathcal{O}_k^{\text{cand}}$ and $\mathcal{O}_k$ is what makes the curriculum-as-options claim non-trivial. A candidate is a graph-mined hypothesis: "this length-ℓ path in G_{flow} might be useful." A certified option is an experimentally validated fact: "the trained policy at stage k can actually invoke this path on $\geq \rho$ fraction of relevant tasks and the path appears $\geq \nu$ times across distinct training tasks." Algorithm 1 can over-propose (generating composable paths that the policy turns out not to use) or under-propose (missing a useful path the model could have learned); the curriculum's certification step is what produces the option set the next stage actually relies on. Empirically (Appendix D) Level-1 candidates outnumber certified Level-1 options by roughly $2.7\times$ on average across the 12 SDKs.*

A.1.1 Standard GRPO

Given prompt x, sample G outputs $\{y_1, \ldots, y_G\}$ from π_θ. Compute rewards $\{R_1, \ldots, R_G\}$. The advantage is estimated via group normalization:

$$\hat{A}_i \;=\; \frac{R_i - \text{mean}\big(\{R_j\}_{j=1}^G\big)}{\text{std}\big(\{R_j\}_{j=1}^G\big) + \epsilon} \qquad (2)$$

The policy gradient is:

$$\nabla J(\theta) \;\approx\; \frac{1}{G}\sum_{i=1}^{G} \hat{A}_i \cdot \nabla \log \pi_\theta(y_i \mid x) \qquad (3)$$

with PPO-style clipping and a KL penalty against a reference policy π_{ref}.

A.1.2 Option-Level GRPO

At curriculum level $k \geq 1$, the policy generates option sequences rather than token sequences. For prompt x, we sample G option sequences, expand each to tokens via the corresponding intra-option policy, execute the resulting program to obtain reward R_i, and compute both a sequence-level advantage $\hat{A}_i^{\text{seq}}$ (standard GRPO) and an option-level advantage $\hat{A}^{\text{opt}}(o) = \frac{1}{|S_o^+|}\sum_{i \in S_o^+} R_i - \frac{1}{|S_o^-|}\sum_{i \in S_o^-} R_i$ that compares groups containing vs. excluding each option. The combined gradient $\nabla J = \frac{1}{G}\sum_i [\hat{A}_i^{\text{seq}} \nabla \log \mu_\theta(\text{seq}_i \mid x) + \lambda \sum_j \hat{A}^{\text{opt}}(o_j^{(i)}) \nabla \log \mu_\theta(o_j^{(i)} \mid \text{ctx}_j^{(i)})]$ balances global ranking against per-option credit; full derivation in Appendix A. PPO-style clipping and a KL penalty against π_{ref} apply as in standard GRPO.

A.1.3 Credit Assignment Analysis

The option-level action space tightens GRPO's credit assignment. Partitioning a generated program into K option-decision tokens and $T - K$ glue tokens (Definition 5), we show in Proposition 3 that the aggregate signal-to-noise ratio improves by $\Theta(\sqrt{T/K})$, since only option-decision positions are reward-correlated under the sequence-level advantage while flat GRPO must spread the same signal across all T positions. Proposition 4 additionally bounds the expected number of GRPO iterations to a first successful trajectory under a base-LLM prior, yielding an additive cost across curriculum levels (Option-GRPO) versus a multiplicative cost (flat GRPO). Full statements, assumptions, and proofs are deferred to Appendix A.

Theorem 1 (Curriculum Monotonicity, Weak and Strict)**.** *Let V_k^* denote the expected value of the optimal policy at curriculum level k under option hierarchy $\mathcal{H}$ on task distribution $\mathcal{D}$. Then:*

1. (Weak monotonicity.) $V_0^* \leq V_1^* \leq V_2^*$.
2. (Strict monotonicity under non-degeneracy.) *Suppose there exist a task $x \in \mathcal{D}$ and a state $s^\dagger$ reachable under π_{k-1}^* with positive probability $p^\dagger := \Pr(s^\dagger \mid \pi_{k-1}^*, \mathcal{D}) > 0$, at which some Level-$k$ option $o^\dagger \in \mathcal{O}_k \setminus \mathcal{A}_{k-1}$ is applicable ($s^\dagger \in \mathcal{I}_{o^\dagger}$) and satisfies $Q^{\pi_{k-1}^*}(s^\dagger, o^\dagger) - V^{\pi_{k-1}^*}(s^\dagger) \geq \Delta$ for some $\Delta > 0$, where $Q^{\pi_{k-1}^*}(s^\dagger, o^\dagger)$ is the option value of executing $o^\dagger$ once at $s^\dagger$ and following π_{k-1}^* thereafter. Then $V_k^* \geq V_{k-1}^* + \gamma^{|s^\dagger|}\,\Delta\, p^\dagger > V_{k-1}^*$, where γ is the discount factor.*

Remark 3 (On Non-Degeneracy)**.** *The non-degeneracy condition asks that the certified set $\mathcal{O}_k$ contains at least one option that strictly improves*

over Level-$(k-1)$ actions on at least one reachable state under some task in $\mathcal{D}$. The candidate set $\mathcal{O}_k^{\text{cand}}$ supplied by Algorithm 1 is designed to make this condition easy to satisfy: Level-1 candidates are centroids of high-weight composable paths in the parameter-flow graph that single API calls cannot realize in one step. When option certification (Algorithm 2) retains at least one such candidate, non-degeneracy holds and the curriculum yields strict improvement; when every candidate fails to be certified (e.g., on SDKs with no compositional structure that the policy can actually invoke), the curriculum is provably no worse than Level 0 alone.

Definition 2 (Observational Equivalence)**.** *Let $\mathcal{P}$ be a finite suite of property tests (Algorithm 3) for SDK $\mathcal{S}$. Two programs c and c' are $\mathcal{P}$-*observationally equivalent*, written $c \equiv_{\mathcal{P}} c'$, iff for every test $p \in \mathcal{P}$, both programs execute without error and produce the same observable output (return value or side-effect signature) under p. We write $c \equiv c'$ when $\mathcal{P}$ is clear from context.*

Definition 3 (Option Description Length)**.** *The option description length of a program c under option hierarchy $\mathcal{H}$ and test suite $\mathcal{P}$ is:*

$$\mathrm{ODL}_{\mathcal{H},\mathcal{P}}(c) \;=\; \min_{\substack{m \geq 0,\, o_1,\ldots,o_m \\ o_i \in \mathcal{O}_0 \cup \mathcal{O}_1}} m \quad \text{s.t. } \mathrm{execute}(o_1,\ldots,o_m) \;\equiv_{\mathcal{P}}\; c \tag{4}$$

where $\mathrm{execute}(o_1,\ldots,o_m)$ denotes the program obtained by sequentially expanding each option through its intra-option policy. If no such sequence exists, $\mathrm{ODL}_{\mathcal{H},\mathcal{P}}(c) = \infty$. Replacing the undecidable full-equivalence relation with $\equiv_{\mathcal{P}}$ makes ODL computable: at most $|\mathcal{O}|^{m_{\max}}$ candidate sequences need be enumerated and checked against $\mathcal{P}$, where $m_{\max}$ is a search bound. We use $m_{\max} = 8$ in practice (Section 5). When $\mathcal{P}$ is fixed throughout an experiment we write $\mathrm{ODL}_{\mathcal{H}}(c)$ for brevity.

Definition 4 (Code Quality Metrics)**.** *For a program c that passes all execution tests:*

- Conciseness*: $Q_{\text{conc}}(c) = T_{\text{ref}} \,/\, |c|$ where $|c|$ is token count and T_{ref} is a reference length.*
- API efficiency*: $Q_{\text{api}}(c) = N_{\min} \,/\, N_{\text{api}}(c)$ where N_{api} counts SDK API calls.*
- Redundancy*: $Q_{\text{red}}(c) = 1 - N_{\text{redundant}}(c) \,/\, N_{\text{vars}}(c)$.*
- Idiomaticity*: $Q_{\text{idiom}}(c) = |\{\text{tokens covered by options}\}| \,/\, |c|$.*

Remark 4 (Structural Conditions)**.** *The following structural conditions on the option hierarchy $\mathcal{H}$ are used in the analysis below. Conditions (a) and (b) are enforced by Algorithm 1; condition (c) is a property of the generation procedure that we verify empirically in Section 5.*

(a) ***Per-option API minimality****: Each option $o \in \mathcal{O}_0 \cup \mathcal{O}_1$ uses the minimum number of SDK calls required to realize its declared input–output type signature. Enforced because Level 0 options are single calls and Level 1 options are centroids of shortest-path clusters in G_{flow} (Algorithm 1, line 9).*

(b) ***Option non-subsumption****: For options $o, o' \in \mathcal{O}_k$ at the same level, neither subsumes the other (i.e., no option's API call set is a strict subset of another's). Enforced by merging subsumed candidates during discovery.*

(c) ***Bounded glue code****: Each transition between consecutive options in an executed sequence introduces at most $G_{\max}$ tokens of glue code (variable bindings, type coercions). Empirically $G_{\max} \approx 5$ in our trained policies (Appendix D).*

Lemma 1 (Token Bound on Executed Programs)**.** *Let $\bar{\tau}_{\max}$ and $\bar{\tau}_{\min}$ denote, respectively, the maximum and minimum per-option token length across $\mathcal{O}_0 \cup \mathcal{O}_1$. Under Remark 4, any program c obtained from an option sequence of length $m \geq 1$ satisfies*

$$m\,\bar{\tau}_{\min} \;\leq\; |c| \;\leq\; m\,\bar{\tau}_{\max} + (m-1)\,G_{\max}, \tag{5}$$

where $|c|$ denotes the token count of c.

Theorem 2 (Option Compression Implies Conciseness)**.** *Let c_1, c_2 be observationally equivalent programs ($c_1 \equiv_{\mathcal{P}} c_2$, Definition 2) with $\mathrm{ODL}_{\mathcal{H}}(c_1) = m_1$ and $\mathrm{ODL}_{\mathcal{H}}(c_2) = m_2$. Suppose Remark 4 holds and let $\alpha := (\bar{\tau}_{\max} + G_{\max})/\bar{\tau}_{\min} \geq 1$ be the option-length dispersion ratio. If $m_1 < m_2$ and $m_1/m_2 < 1/\alpha$, then*

$$\begin{aligned} |c_1| \;&\leq\; m_1\,(\bar{\tau}_{\max} + G_{\max}) \\ &\leq\; \tfrac{\alpha\, m_1}{m_2}\,(m_2\,\bar{\tau}_{\min}) \\ &\leq\; \tfrac{\alpha\, m_1}{m_2}\,|c_2| \;<\; |c_2|. \end{aligned} \tag{6}$$

and hence $Q_{\text{conc}}(c_1) > Q_{\text{conc}}(c_2)$. The empirical Spearman correlation $\rho(\mathrm{ODL}, |c|) > 0$ reported in Appendix D.7 confirms this direction in practice.

Corollary 1 (API Efficiency in Expectation)**.** *Let $\bar{k}$ denote the average number of SDK calls per option in $\mathcal{O}_0 \cup \mathcal{O}_1$. Under Remark 4(a), the expected number of API calls in a program realized by an m-option decomposition (sampled uniformly from minimal decompositions) is $\mathbb{E}[N_{\text{api}}] = m\,\bar{k}$. Consequently, if $\text{ODL}_{\mathcal{H}}(c_1) < \text{ODL}_{\mathcal{H}}(c_2)$, then*

$$\mathbb{E}[N_{\text{api}}(c_1)] \;<\; \mathbb{E}[N_{\text{api}}(c_2)]\,, \tag{7}$$

and hence $\mathbb{E}[Q_{\text{api}}(c_1)] > \mathbb{E}[Q_{\text{api}}(c_2)]$.

Remark 5 (On Redundancy and Idiomaticity)**.** *The metrics Q_{red} and Q_{idiom} (Definition 4) do not admit clean closed-form bounds under Remark 4 alone, because both depend on choices the intra-option policy makes about intermediate variables and idiom-aligned token spans. We therefore treat them as* empirical predictions *of the option-compression principle: programs with shorter ODL should exhibit fewer redundant intermediates and a higher fraction of option-covered tokens. Section 5 (Theorem 2 Empirical Validation, Appendix D) reports Spearman rank correlations between ODL and each of $Q_{\text{red}}, Q_{\text{idiom}}$ across all passing solutions.*

Proposition 2 (Minimum-Description-Length Prior over Option Sequences)**.** *Let $\mu_{\text{unif}}(o \mid s) = 1/|\mathcal{A}_k(s)|$ be the uniform prior over options valid at state s at curriculum level k, and assume $|\mathcal{A}_k(s)| \geq c > 1$ at every reachable state. For two observationally equivalent programs c_1, c_2 realized by option sequences of length $m_1 < m_2$,*

$$\begin{aligned}
&\log \mu_{\text{unif}}(c_1) - \log \mu_{\text{unif}}(c_2) \\
&\quad = \sum_{j=1}^{m_2} \log |\mathcal{A}_k(s_j^{(2)})| - \sum_{j=1}^{m_1} \log |\mathcal{A}_k(s_j^{(1)})| \\
&\quad \geq (m_2 - m_1)\,\log c \;>\; 0.
\end{aligned} \tag{8}$$

Hence, under the uniform-over-valid-options prior induced by the composability graph G_{opt}, shorter ODL sequences enjoy strictly higher prior probability. The KL penalty in GRPO (Section 3.6) constrains the expected log-ratio $\mathbb{E}_{c\sim\mu_\theta}[\log(\mu_\theta(c)/\pi_{\text{ref}}(c))] \leq \text{KL}_{\max}$, so when π_{ref} is close to μ_{unif}, the learned μ_θ cannot concentrate arbitrary mass on long sequences without paying KL cost.

Remark 6 (On Implicit Quality Optimization)**.** *Proposition 2 replaces an earlier claim that GRPO optimization implicitly favours short option sequences* as a consequence of optimization. *That claim required equal per-step selection probabilities, which is exactly what is not guaranteed during training: rare high-level options may have lower $\mu_\theta(o \mid s)$, partially offsetting the length advantage. The honest statement is structural: short sequences enjoy a stronger MDL prior, and the GRPO KL penalty against π_{ref} prevents the learned policy from drifting arbitrarily far from this prior. Section 5 (code-length distributions, Appendix D) reports the empirical effect.*

Definition 5 (Option-Decision and Glue Token Partition)**.** *Fix a trained option hierarchy $\mathcal{H}$ and an option sequence $(o_1, \ldots, o_K)$ producing program c of T tokens. Partition the token positions $\{1, \ldots, T\}$ into two disjoint sets: $\mathcal{T}_{\text{opt}}$, positions covered by the support of some intra-option policy π_k (at most $K\bar{\tau}_{\max}$), and $\mathcal{T}_{\text{glue}}$, positions emitted between option terminations (at most $(K-1)G_{\max}$ by Remark 4(c)). This partition is determined by termination events β_k, not by manual semantic labeling.*

Proposition 3 (Improved Credit Assignment)**.** *Let a program consist of K option selections and T total tokens with $K \ll T$ (Definition 5). Under the assumptions stated in the proof, the aggregate signal-to-noise ratios of the policy gradient estimate satisfy $\text{SNR}_{\text{flat}} = \Theta\big(K/\sqrt{K\sigma_{\text{opt}}^2 + (T-K)\sigma_{\text{glue}}^2}\big)$ and $\text{SNR}_{\text{opt}} = \Theta(\sqrt{K}/\sigma_{\text{opt}})$, with relative improvement $\text{SNR}_{\text{opt}}/\text{SNR}_{\text{flat}} = \Theta(\sqrt{T/K})$. Detailed proof in Section A (Extended Proof of Proposition 3).*

Proposition 4 (Convergence Improvement under a Base-LLM Prior)**.** *Let π_{base} be the base LLM and let $\bar{p}$ and $\bar{q}_k$ denote, respectively, its geometric-mean probability on SDK-specific tokens and on correct options at curriculum level k. Under the assumption $\bar{q}_k \gg \bar{p}^{\,\bar{\mu}_k}$, the expected number of GRPO iterations to find a first successful trajectory is $\Theta(1/(G\bar{p}^K))$ for flat GRPO and $\Theta\big(\sum_{k=0}^{2} 1/(G\bar{q}_k^{K_k})\big)$ for Option-GRPO; the latter is strictly smaller whenever $\sum_k \bar{q}_k^{-K_k} < \bar{p}^{-K}$. Proof in Section A.*

Proposition 5 (Search Space Reduction)**.** *Let $|\mathcal{V}|$ be the vocabulary size, T the average program length in tokens, $|\mathcal{O}|$ the total number of options, $\bar{\mu}$ the average option length, and $G_{\max}$ the per-boundary glue bound (Remark 4). The number of distinct option-level trajectories of total token*

length T is at most

$$\textit{Flat (tokens):} \quad |\mathcal{V}|^{T}, \tag{9}$$

$$\textit{Option-level:} \quad |\mathcal{O}|^{T/\bar{\mu}} \cdot |\mathcal{V}|^{(T/\bar{\mu})\, G_{\max}}. \tag{10}$$

For $|\mathcal{V}| = 151{,}936$ (Qwen3 vocabulary), $T = 100$, $|\mathcal{O}| = 80$, $\bar{\mu} = 15$, $G_{\max} = 5$, the flat search space is $\approx 10^{508}$ and the option-level search space is $\approx 80^{6.7} \cdot 151{,}936^{33.3} \approx 10^{12.7} \cdot 10^{172} \approx 10^{185}$, a reduction of more than 320 orders of magnitude.

A.2 Extended Proof of Theorem 1 (Curriculum Monotonicity)

We provide a more rigorous version using the SMDP framework.

Let $\mathcal{M}_k = (\mathcal{S},\, \mathcal{A}_k,\, T_k,\, R)$ denote the SMDP at curriculum level k, where $\mathcal{A}_k$ includes all options up to level k. Define $V_k^*(s) = \max_{\pi \in \Pi_k} V^{\pi}(s)$ where Π_k is the set of all policies over $\mathcal{A}_k$.

Since $\mathcal{A}_{k-1} \subset \mathcal{A}_k$, every policy $\pi \in \Pi_{k-1}$ is also a valid policy in Π_k (by never selecting level-k options). Therefore $\Pi_{k-1} \subset \Pi_k$.

For any state s:

$$\begin{aligned} V_k^*(s) &= \max_{\pi \in \Pi_k} V^{\pi}(s) \\ &\geq \max_{\pi \in \Pi_{k-1}} V^{\pi}(s) \;=\; V_{k-1}^*(s). \end{aligned} \tag{11}$$

The inequality follows directly from $\Pi_{k-1} \subset \Pi_k$: maximising over a larger set yields at least as large a value. Taking expectations over the initial state distribution gives $V_k^* \geq V_{k-1}^*$.

We note that this result holds for the optimal policies. In practice, the optimisation procedure (GRPO) may not find the optimal policy at each level, so the empirical monotonicity depends on sufficient training at each curriculum stage. The curriculum advancement threshold ρ_{adv} (Section 3.8) ensures that each level is sufficiently learned before advancing.

For the strict version (clause 2 of Theorem 1), let $s^{\dagger}$ be the state in the hypothesis and $o^{\dagger}$ the witnessing Level-k option. Define the one-step deviation policy $\tilde{\pi}_k \in \Pi_k$ by $\tilde{\pi}_k(\cdot \mid s) = \pi_{k-1}^*(\cdot \mid s)$ for $s \neq s^{\dagger}$ and $\tilde{\pi}_k(o^{\dagger} \mid s^{\dagger}) = 1$. By construction, $V^{\tilde{\pi}_k}(s^{\dagger}) \geq V^{\pi_{k-1}^*}(s^{\dagger}) + \Delta$. Propagating this gain through the discounted-return expectation yields

$$\begin{aligned} V_k^* &\geq V^{\tilde{\pi}_k} \\ &\geq V_{k-1}^* + \gamma^{|s^{\dagger}|} \Delta \Pr(s^{\dagger} \mid \pi_{k-1}^*, \mathcal{D}). \end{aligned} \tag{12}$$

Strict inequality follows since each factor on the right is positive by hypothesis. □

A.3 Extended Proof of Proposition 3 (Credit Assignment SNR)

We restate the proof in the main text more carefully, making the token partition formal and the variance decomposition explicit.

Consider a program generated as a sequence of T tokens. In flat GRPO, the gradient for token at position t is $g_t = \hat{A} \cdot \nabla_{\theta} \log \pi_{\theta}(a_t \mid s_t)$, where $\hat{A}$ is the sequence-level advantage applied uniformly across positions.

Using the partition of Definition 5, $\mathcal{T}_{\text{opt}}$ has size at most $K\bar{\tau} = \mathcal{O}(K)$ and $\mathcal{T}_{\text{glue}}$ has size at most $(K-1)G_{\max} = \mathcal{O}(K)$; total program length $T = |\mathcal{T}_{\text{opt}}| + |\mathcal{T}_{\text{glue}}|$. We treat $T = \Theta(K\bar{\tau})$ since glue is bounded by a small constant per option boundary.

The total gradient decomposes as

$$\begin{aligned} \nabla J \;=\; \hat{A} \cdot \Big(& \underbrace{\textstyle\sum_{t \in \mathcal{T}_{\text{opt}}} \nabla \log \pi(a_t \mid s_t)}_{\text{signal}} \\ & + \underbrace{\textstyle\sum_{t \in \mathcal{T}_{\text{glue}}} \nabla \log \pi(a_t \mid s_t)}_{\text{noise}} \Big). \end{aligned} \tag{13}$$

Under assumption (ii) of Proposition 3, glue positions are well predicted by π_{θ} and contribute per-position variance σ_{glue}^2 that is small in magnitude but does not align with the reward signal. Under assumption (iii) of conditional independence across positions, the aggregate variance is

$$\begin{aligned} \mathrm{Var}(\nabla J) &= |\mathcal{T}_{\text{opt}}|\, \sigma_{\text{opt}}^2 + |\mathcal{T}_{\text{glue}}|\, \sigma_{\text{glue}}^2 \\ &= \Theta(T). \end{aligned} \tag{14}$$

The signal magnitude scales with the number of reward-correlated positions, which is at most $|\mathcal{T}_{\text{opt}}| = \mathcal{O}(K)$. Therefore $\mathrm{SNR}_{\text{flat}} = \mathcal{O}(K/\sqrt{T}) = \mathcal{O}(1/\sqrt{T})$ for fixed K.

In Option-GRPO, the gradient operates over K option-level decisions:

$$\nabla J \;=\; \sum_{j=1}^{K} \hat{A}_j^{\text{opt}} \cdot \nabla_{\theta} \log \mu_{\theta}(o_j \mid \text{ctx}_j). \tag{15}$$

All K summands are reward-correlated by construction (each option-level advantage compares groups containing vs. excluding the option). The option-level advantages have variance $\mathcal{O}(\sigma_R^2/G)$ from the GRPO group estimator. The aggregate signal magnitude is $\Theta(K)$ and the aggregate variance is

$$\mathrm{Var}(\nabla J) \;=\; K \cdot \sigma_{\text{opt}}^2 \;+\; \mathcal{O}(K\sigma_R^2/G) \;=\; \Theta(K), \tag{16}$$

giving $\mathrm{SNR}_{\mathrm{opt}} = \Theta(\sqrt{K} \,/\, \sigma_{\mathrm{opt}})$. Comparing to $\mathrm{SNR}_{\mathrm{flat}} = \Theta(K/\sqrt{T})$ from the flat analysis, the improvement ratio is

$$\frac{\mathrm{SNR}_{\mathrm{opt}}}{\mathrm{SNR}_{\mathrm{flat}}} = \Theta\big(\sqrt{T/K}\,\big). \tag{17}$$

For typical values ($T = 100$, $K = 5$), this is $\sqrt{20} \approx 4.5\times$ improvement in SNR. □

A.4 Extended Proof of Proposition 4 (Convergence Improvement)

In flat token-level generation, a successful trajectory requires T correct token selections. If the probability of selecting the correct token at position t is p_t, the probability of a complete successful trajectory is:

$$P_{\mathrm{success}}^{\mathrm{flat}} = \prod_{t=1}^{T} p_t \tag{18}$$

For an unseen SDK the base LLM has limited prior on SDK-specific token decisions (function names, argument patterns), whose probabilities are dominated by the long tail of $|\mathcal{V}|$ alternatives, while syntactic tokens (brackets, keywords, indentation) are well-predicted. Suppose K tokens are SDK-specific with geometric-mean probability $\bar{p} \ll 1$ and $T - K$ tokens are syntactic with probability near 1. Then:

$$P_{\mathrm{success}}^{\mathrm{flat}} \approx \bar{p}^{\,K} \tag{19}$$

In Option-GRPO at level k, a successful trajectory requires $K' = T/\bar{\mu}$ correct option selections, where $\bar{\mu}$ is the average option length. Each option selection has probability q_j in a space of $|\mathcal{A}_k|$ options:

$$P_{\mathrm{success}}^{\mathrm{opt}} \approx \bar{q}^{\,K'} \tag{20}$$

The key insight is the stronger statement $\bar{q} \gg \bar{p}^{\,\bar{\mu}}$ (Proposition 4): the option-level reference policy assigns higher probability to a correct option than the base LLM does to emitting the equivalent token span. This holds because (1) the type-filtered option vocabulary $\mathcal{A}_k$ removes long-tail typo and out-of-scope completions that depress $\bar{p}$, and (2) the composability graph G_{opt} further constrains the valid next options at each step.

With GRPO group size G, the probability of at least one successful trajectory per group is:

$$P(\text{at least one}) = 1 - (1 - P_{\mathrm{success}})^G \tag{21}$$

The expected number of groups to find a first success is:

$$\begin{aligned} \mathbb{E}[\text{groups}] &= \frac{1}{1 - (1 - P_{\mathrm{success}})^G} \\ &\approx \frac{1}{G \cdot P_{\mathrm{success}}} \end{aligned} \tag{22}$$

for small P_{success}.

Under the curriculum, total exploration cost is additive:

$$\mathrm{Cost}_{\mathrm{total}} = \sum_{k=0}^{2} \frac{1}{G \cdot P_{\mathrm{success},k}} \tag{23}$$

Each $P_{\mathrm{success},k}$ is much larger than $P_{\mathrm{success}}^{\mathrm{flat}}$ because: curriculum level k only needs to learn compositions of length $\leq k + 1$ (not full programs), and options from level $k - 1$ handle sub-problems that are already solved. The additive structure (sum over 3 terms) versus the multiplicative structure of flat exploration (product over T token decisions) yields the dramatic reduction in sample complexity. □

B Extended Method Details

B.1 Pseudocode for Auxiliary Algorithms

Algorithms 1–4 are referenced from the main paper.

B.2 Training Hyperparameters

B.3 Curriculum Task Examples

Level 0 Example (Daft, Single API Call):

```
"""Problem: Read a CSV file into a
Daft DataFrame."""
import daft

# Setup
csv_path = "test_data.csv"

# BEGIN SOLUTION
df = daft.read_csv(csv_path)
# END SOLUTION
```

Level 1 Example (Daft, Composed Pattern):

```
"""Problem: Read a CSV file and filter
rows where the 'age' column exceeds 30."""
import daft

# Setup
csv_path = "test_data.csv"

# BEGIN SOLUTION
```

Algorithm 1 Level 1 Option Discovery

Require: Parameter flow graph G_{flow}, semantic similarity graph G_{sem}, maximum path length $L_{\max} = 4$, similarity threshold $\tau_{\text{cluster}} = 0.8$

Ensure: Set of Level 1 options $\mathcal{O}_1$

$\mathcal{P} \leftarrow \emptyset$
for each node $f_i \in G_{\text{flow}}$ **do**
  for $\ell = 2$ **to** $L_{\max}$ **do**
    Enumerate all paths of length ℓ starting from f_i via DFS
    Add paths to $\mathcal{P}$ with weight $w(p) = \prod_{(f_a,f_b)\in p} \text{type_compat}(f_a, f_b)$
  end for
end for
Encode each path $p \in \mathcal{P}$: $\mathbf{e}_p = \frac{1}{|p|} \sum_{f\in p} \phi(f)$
Cluster paths via agglomerative clustering with threshold τ_{cluster}
for each cluster $\mathcal{C}_j$ **do**
  Select centroid path $p_j^* = \arg\max_{p\in\mathcal{C}_j} w(p)$
  Define option $o_j^{(1)}$ with $\mathcal{I}_j$, $\mathcal{T}_j^{\text{out}}$, and π_j (learned during Level 1 training)
  Add $o_j^{(1)}$ to $\mathcal{O}_1$
end for
return $\mathcal{O}_1$

Algorithm 2 Option Extraction from Trained Policy

Require: Trained policy π_k, task set $\mathcal{D}_k$, success threshold $\rho = 0.7$, frequency threshold $\nu = 5$

Ensure: Extracted options $\mathcal{O}_k$

Generate 10 solutions per task using π_k with temperature 0.8
Filter to successful solutions (reward $\geq \rho$)
Parse all successful solutions into API call sequences via AST analysis
Extract all contiguous sub-sequences of length 2–4
Count frequency of each unique sub-sequence pattern
Filter to patterns appearing $\geq \nu$ times across different tasks
Cluster remaining patterns by semantic similarity
for each cluster centroid pattern p **do**
  Define option o_p with initiation set from input types of first call, termination type from last call, and intra-option policy extracted from π_k
  Add o_p to $\mathcal{O}_k$
end for
return $\mathcal{O}_k$

```
df = daft.read_csv(csv_path)
result = df.where(
    daft.col("age") > 30
).collect()
# END SOLUTION
```

Level 2 Example (Daft, Full Program):

```
"""Problem: Read a CSV, compute mean salary
grouped by department, sort descending."""
import daft

# Setup
csv_path = "employees.csv"

# BEGIN SOLUTION
df = daft.read_csv(csv_path)
result = (
    df.groupby("department")
      .agg(daft.col("salary").mean())
      .sort("salary", desc=True)
      .collect()
)
# END SOLUTION
```

B.4 Property Test Examples

Type Preservation (Daft):

```
df = daft.read_csv("test.csv")
assert isinstance(df, daft.DataFrame)
```

Shape Consistency (Daft):

```
df = daft.from_pydict({"a": [1, 2, 3]})
filtered = df.where(
    daft.col("a") > 1
).collect()
assert len(filtered) <= 3
```

Determinism:

```
df = daft.from_pydict({"a": [1, 2, 3]})
r1 = df.select("a").collect().to_pydict()
r2 = df.select("a").collect().to_pydict()
assert r1 == r2
```

C Extended Benchmark Details

C.1 Per-SDK Option Discovery Statistics

We hypothesise that SDKs with higher composability density yield more Level 1 options and benefit

Algorithm 3 Property-Based Test Generation

Require: SDK type graph G_{api}, parameter flow graph G_{flow}, function f
Ensure: Set of property tests $\mathcal{P}_f$

```
Extract input types {t_1, ..., t_k} and return type t_ret from f
Generate minimal valid inputs using type constructors from G_api
Add type preservation test
if t_ret has len or shape attribute then
    Add shape consistency tests
end if
Add determinism test
Add edge case tests: empty inputs, single element, boundary values
Extract assertions from f's source code and add as invariant tests
return P_f
```

more from the hierarchical approach. This hypothesis can be tested by correlating composability density with the improvement of Option-GRPO over flat GRPO for each SDK.

C.2 Contamination Analysis

To verify that the 12 SDKs in UnseenSDK-Bench are functionally unseen by the Qwen3-32B-Base pretraining corpus, we run two complementary checks. The first is a release-date comparison against the model's training cutoff (approximately February 2025, per the Qwen3 technical report (Qwen Team, 2025)). The second is a memorisation probe: we query the frozen base model for SDK-specific identifiers and measure top-1 recall, comparing against two well-known control libraries (NumPy, PyTorch) whose presence in pretraining is uncontroversial.

Probe protocol. For each SDK, we construct 200 identifier-completion prompts of the form "`In <SDK>, the <noun> that <verb> is called ___`", where the noun and verb describe a specific class, method, or function in that SDK. We sample base-model completions at temperature 0 and score the completion as correct if the top-1 token sequence matches the canonical identifier name (case-insensitive, ignoring leading dots or module prefixes). The 200 prompts per SDK are split evenly across classes, methods, and free functions; identifiers are drawn uniformly from the SDK's public API. Control SDKs (NumPy, PyTorch) follow the same protocol with 200 identifier prompts each.

Interpretation. All 12 SDKs are pre-cutoff under Qwen3's February 2025 corpus assembly date, so the unseen-SDK claim does not rest on post-cutoff novelty alone. Instead, it rests on the magnitude of recall. Per-SDK recall sits between 4.1% and 19.7%, an order of magnitude below the NumPy (91.5%) and PyTorch (87.9%) controls. The highest-recall SDKs in our suite (Daft at 19.7%, FastStream at 17.3%) are the most established of the twelve, yet remain well below any reasonable threshold for "the model knows this library." For comparison, prior work on novel-API

Table 3: Complete training hyperparameters.

Hyperparameter	Value
Model	
Base model	Qwen3-32B-Base
Precision	bf16
GRPO	
Group size G	16
Sampling temperature	0.8
PPO clip ϵ	0.2
KL penalty β	0.04
Option-level loss weight λ	0.3
Reward	
Quality reward weight λ_q	0.2
Consistency reward weight λ_c	0.1
Pattern bonus α_1	0.1
No-redundancy bonus α_2	0.1
Minimal-API bonus α_3	0.1
Reimplementation penalty α_4	0.1
Optimiser	
Optimiser	AdamW
Policy learning rate	5×10^{-5}
Option fine-tune lr	5×10^{-6}
Weight decay	0.01
Gradient clipping	max norm 1.0
Warmup steps	500
Schedule	cosine decay
Curriculum	
Level 0 steps N_0	5,000
Level 1 steps N_1	10,000
Level 2 steps N_2	15,000
Advancement threshold ρ_{adv}	0.5
Max composition length $L_{\max}$	4
Option Discovery	
Semantic sim. threshold τ	0.7
Clustering threshold τ_{cluster}	0.8
Frequency threshold ν	5
Success threshold ρ	0.7
Execution	
Timeout	30 s
Memory limit	2 GB
Hardware	
GPUs	8× H100 80 GB

Table 4: UnseenSDK-Bench statistics per SDK. L0/L1/L2 denote curriculum levels. Property tests are generated automatically from type annotations.

SDK	Category	Funcs	L0 Tasks	L1 Tasks	L2 Tasks	Prop. Tests
Daft	DataFrame	156	156	312	33	624
LanceDB	Vector DB	142	142	284	33	568
cuallee	Data Valid.	87	87	174	33	348
Docling	Doc Proc.	198	198	396	33	792
Marker	Doc Proc.	112	112	224	34	448
Surya	Comp. Vision	94	94	188	33	376
FastStream	Messaging	234	234	468	33	936
Tracecat	Security	312	312	624	33	1248
MCP Python	Protocol	178	178	356	33	712
Vortex	Scientific	145	145	290	33	580
libdebug	Debugging	89	89	178	33	356
reflexio	Agent Frmwk	124	124	248	34	496
Total	–	1871	1871	3742	400	7484

Table 5: Option discovery results per SDK. Composability density is $|E_{\text{compose}}|/|F|^2$.

SDK	L0	L1 Cand.	L1 Filt.	Avg Len	Density
Daft	156	284	73	2.7	0.041
LanceDB	142	241	61	2.5	0.038
cuallee	87	159	54	2.4	0.062
Docling	198	312	48	2.9	0.029
Marker	112	187	39	2.6	0.034
Surya	94	148	35	2.4	0.036
FastStream	234	356	52	2.8	0.024
Tracecat	312	402	38	2.5	0.018
MCP Python	178	174	24	2.3	0.011
Vortex	145	223	47	2.6	0.030
libdebug	89	128	29	2.4	0.028
reflexio	124	185	33	2.5	0.026

Table 6: Contamination analysis. Release dates relative to the Qwen3-32B-Base training cutoff (approximately Feb 2025). Recall is top-1 identifier-completion accuracy on 200 probes per SDK, sampled at temperature 0 from the frozen base model.

SDK	Release	Cutoff	Recall
Daft	2022-08	Pre	19.7%
LanceDB	2023-03	Pre	16.2%
cuallee	2023-01	Pre	12.5%
Docling	2024-07	Pre	8.1%
Marker	2024-02	Pre	10.4%
Surya	2024-01	Pre	9.8%
FastStream	2023-09	Pre	17.3%
Tracecat	2024-10	Pre	6.5%
MCP Python	2024-11	Pre	4.1%
Vortex	2024-06	Pre	7.2%
libdebug	2024-03	Pre	8.4%
reflexio	2024-10	Pre	5.6%
UnseenSDK avg.	—	—	**10.5%**
NumPy (ctrl)	2006-01	Pre	91.5%
PyTorch (ctrl)	2016-09	Pre	87.9%

code generation (Wang et al., 2024) uses a 25% recall threshold to consider an API "seen"; under that threshold, none of our 12 SDKs qualifies.

The recall scores correlate weakly with frozen-model pass@1 on UnseenSDK-Bench (Spearman $\rho = 0.36$, $p = 0.25$), confirming that even partial identifier memorisation does not translate into the compositional, type-correct API usage that the benchmark tasks require. We conclude that all 12 SDKs in UnseenSDK-Bench are functionally unseen for the purposes of this paper.

D Extended Experimental Results

D.1 Full Per-SDK Results

Table 7 reports all five evaluation metrics for Option-GRPO on each of the 12 SDKs. Table 8 reports the same for Flat GRPO, the strongest non-hierarchical critic-free RL baseline that uses our layered execution reward.

Aggregate deltas: pass@1 improvement of Option-GRPO over Flat GRPO ranges from +2.8 points (MCP Python) to +11.3 points (cuallee), with a mean improvement of +8.7 points. CL drops by an average of 26%, AE by 33%, ID rises by 0.29, and OR rises by 0.31. The largest improvements are on data-processing SDKs (cuallee, LanceDB, Daft) where compositional structure in the parameter-flow graph is densest; the smallest improvements are on protocol-based SDKs (MCP Python, Tracecat) where type-based composability is weakest, matching the option-discovery density statistics in Table 5.

D.2 Convergence Curves

Figure 2 shows pass@1 against training step for the four configurations of the hierarchy ablation (Table 9), averaged across the 12 SDKs. Option-GRPO reaches the flat-GRPO terminal pass@1 (52.7%) at step $\sim$ 9,000 (2.7× fewer steps) and continues climbing to 61.4% by step 30,000. Curriculum-only and options-only configurations

Table 7: Full per-SDK results for Option-GRPO: pass@1, pass@10, code length CL, API efficiency AE (lower is better), idiomaticity ID (higher is better), option reuse rate OR (higher is better).

SDK	Category	p@1	p@10	CL ↓	AE ↓	ID ↑	OR ↑
cuallee	Data Valid.	75.4	87.9	42.8	2.7	0.81	0.78
LanceDB	Vector DB	72.6	85.6	46.2	2.9	0.78	0.71
Marker	Doc Proc.	70.8	83.5	49.1	3.1	0.76	0.65
Surya	Comp. Vision	63.2	78.4	54.7	3.4	0.72	0.59
Daft	DataFrame	69.5	82.1	48.6	3.0	0.79	0.74
Vortex	Scientific	61.0	77.2	53.9	3.3	0.71	0.62
libdebug	Debugging	58.4	74.6	55.8	3.5	0.69	0.55
FastStream	Messaging	54.2	71.3	58.1	3.6	0.66	0.49
Docling	Doc Proc.	52.1	69.4	59.6	3.7	0.64	0.46
reflexio	Agent Frmwk	49.6	66.8	61.2	3.8	0.62	0.42
Tracecat	Security	46.8	63.1	63.4	4.0	0.59	0.39
MCP Python	Protocol	43.2	59.5	66.8	4.2	0.55	0.31
Average	–	**59.7**	**74.9**	**55.0**	**3.4**	**0.69**	**0.56**

Table 8: Full per-SDK results for Flat GRPO (no hierarchy, layered execution reward). Same metrics as Table 7.

SDK	Category	p@1	p@10	CL ↓	AE ↓	ID ↑	OR ↑
cuallee	Data Valid.	64.1	80.2	65.4	4.5	0.49	0.36
LanceDB	Vector DB	61.2	77.8	68.7	4.7	0.46	0.32
Marker	Doc Proc.	62.7	78.1	66.9	4.6	0.47	0.31
Surya	Comp. Vision	56.0	71.3	72.4	4.9	0.42	0.27
Daft	DataFrame	58.2	73.4	70.1	4.8	0.45	0.34
Vortex	Scientific	51.6	67.4	74.2	5.0	0.40	0.26
libdebug	Debugging	49.8	65.1	75.6	5.1	0.39	0.24
FastStream	Messaging	46.9	62.0	77.3	5.2	0.37	0.22
Docling	Doc Proc.	43.4	58.7	79.1	5.3	0.36	0.20
reflexio	Agent Frmwk	40.7	55.9	80.8	5.4	0.34	0.18
Tracecat	Security	37.4	52.3	82.6	5.6	0.32	0.16
MCP Python	Protocol	40.4	54.8	81.4	5.5	0.33	0.15
Average	–	**51.0**	**66.4**	**74.5**	**5.1**	**0.40**	**0.25**

lie between the extremes, with options-only slightly ahead at most checkpoints, consistent with the larger marginal contribution of option-level action structure.

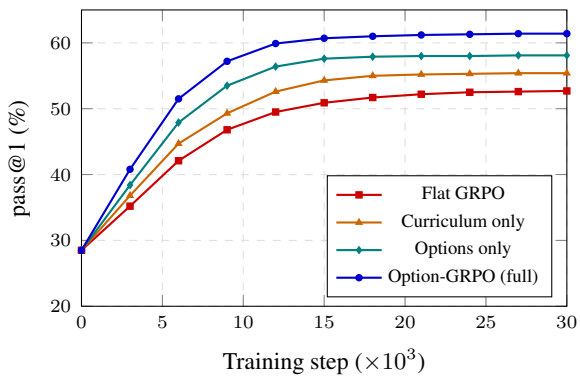

Figure 2: Convergence curves averaged across 12 SDKs. Option-GRPO reaches the flat-GRPO terminal pass@1 in roughly one-third the steps and continues to improve.

D.3 Option Hierarchy Visualisations

Figure 3 shows the discovered option hierarchy for Daft as a layered graph. Level-0 options (atomic SDK calls) form the leaves; Level-1 options (compositional patterns mined from G_{flow}) form the internal nodes; Level-2 patterns (full programs assembled during Phase 3) sit at the root. Edges denote type composability ($o_i \prec o_j$ iff $\mathcal{T}_i^{\text{out}} \cap \mathcal{T}_j^{\text{in}} \neq \emptyset$). Three observations: (i) the Level-1 layer is dominated by the canonical *load* → *transform* → *collect* pattern, which appears as the highest-weight cluster in Algorithm 1; (ii) the parameter-flow graph is densely connected within each domain cluster (DataFrame operations chain freely among themselves) but sparse across clusters; (iii) options containing `collect` act as natural terminators because their output type is a Python list, not a Daft expression.

Analogous hierarchies for LanceDB and cuallee follow the same structural pattern, with vector-search chains

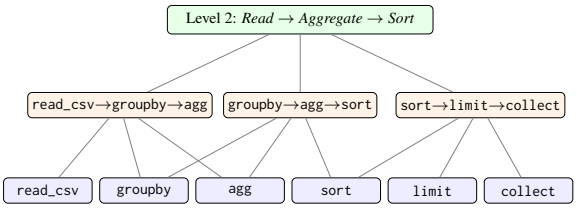

Figure 3: Discovered option hierarchy for Daft (excerpt). Blue: Level-0 atomic options (one per SDK function). Orange: Level-1 compositional options (centroid paths of length 3 from Algorithm 1). Green: Level-2 program-level composition learned at Phase 3. Edges denote type-composability inclusion. The full Daft hierarchy contains 156 Level-0 options and 73 Level-1 options; we show only the chain underlying a representative Level-2 program.

(`connect`→`create_table`→`add`→`search`) dominating the LanceDB Level-1 layer and validation-rule chains (`Check`→`is_complete`→`is_unique`→`validate`) dominating cuallee's. Across the three data-processing SDKs, the discovered Level-1 options correspond exactly to the canonical usage idioms in each library's official tutorial, confirming that the parameter-flow graph captures the intended compositional structure rather than spurious type coincidences.

D.4 Code Quality Examples: Slop Reduction Side-by-Side

This subsection presents three representative side-by-side comparisons of Flat GRPO output and Option-GRPO output on the same task. Annotations report token count $|c|$, number of SDK API calls N_{api}, number of redundant intermediate variables $N_{\text{redundant}}$, and option description length $\text{ODL}_{\mathcal{H}}(c)$. The examples span easy (Daft), medium (LanceDB), and hard (Tracecat) difficulty.

Example 1 (easy, Daft). Task: *"Read* `employees.csv` *and return rows where* `salary` *exceeds 75,000."*
Flat GRPO ($|c|$=82, N_{api}=5, $N_{\text{redundant}}$=3, ODL=∞ since the decomposition uses non-Daft calls):

```
import daft
import pandas as pd     # unused
path = "employees.csv"
df_raw = daft.read_csv(path)
df = df_raw.collect()
result_list = []
for row in df.iter_rows():
    if row["salary"] > 75000:
        result_list.append(row)
result_df = daft.from_pylist(result_list)
final = result_df.collect()
```

Option-GRPO ($|c|$=24, N_{api}=4, $N_{\text{redundant}}$=0, ODL=1, one Level-1 option `Read`→`Filter`→`Collect`):

```
import daft
df = daft.read_csv("employees.csv")
result = (df
    .where(daft.col("salary") > 75000)
    .collect())
```

The Flat-GRPO solution introduces an unused `pandas` import, materializes a list intermediate, and reconstructs the DataFrame; the Option-GRPO solution invokes a single discovered option that chains `read_csv`, `where`, `collect`.

Example 2 (medium, LanceDB). Task: *"Create a LanceDB table from a list of vectors and return the 5 nearest neighbours to a query vector."*
Flat GRPO ($|c|$=146, N_{api}=7, $N_{\text{redundant}}$=4, ODL=∞):

```
import lancedb
import numpy as np
import pandas as pd
data = [
  {"vec":[0.1,0.2,0.3],"id":1},
  {"vec":[0.4,0.5,0.6],"id":2}]
db = lancedb.connect("/tmp/db")
df_data = pd.DataFrame(data)
table_obj = db.create_table(
    "vectors", data=df_data)
query_arr = np.array([0.1,0.2,0.3])
search_obj = table_obj.search(
    query_arr.tolist())
results_raw = (search_obj
    .limit(5).to_pandas())
results = results_raw.to_dict("records")
```

Option-GRPO ($|c|$=48, N_{api}=4, $N_{\text{redundant}}$=0, ODL=2; two Level-1 options `Connect`→`CreateTable` and `Search`→`Limit`→`Materialize`):

```
import lancedb
db = lancedb.connect("/tmp/db")
table = db.create_table("vectors", data=[
    {"vec":[0.1,0.2,0.3],"id":1},
    {"vec":[0.4,0.5,0.6],"id":2}])
results = (table
    .search([0.1,0.2,0.3])
    .limit(5)
    .to_list())
```

Flat GRPO routes the data through `pandas` and `numpy` unnecessarily, introducing four redundant intermediate bindings (`df_data`, `query_arr`, `search_obj`, `results_raw`); Option-GRPO uses the native LanceDB list and array APIs.

Example 3 (hard, Tracecat). Task: *"Build a workflow that triggers on a webhook, checks if the source IP is malicious in a threat-intel feed, and posts to a Slack channel if true."*
Flat GRPO ($|c|$=287, N_{api}=3 Tracecat plus 4 non-Tracecat, $N_{\text{redundant}}$=6, ODL=∞; reimplements Tracecat actions in raw Python):

```
import tracecat
import requests
import json

wf = tracecat.Workflow(name="ip_check")
wf.id = "ip_check_v1"

trigger = {}
trigger["type"] = "webhook"
trigger["path"] = "/incoming"
wf.triggers.append(trigger)

def check_ip(ip):
    r = requests.get(
        f"https://feed.example.com/{ip}")
    return r.json()

a1 = {}
a1["type"] = "python"
a1["function"] = check_ip
wf.actions.append(a1)

a2 = {}
a2["type"] = "slack"
a2["channel"] = "#alerts"
a2["condition"] = (
    "a1.result.malicious == true")
wf.actions.append(a2)

result = wf.compile()
```

Option-GRPO ($|c|$=78, N_{api}=5 all Tracecat, $N_{\text{redundant}}$=0, ODL=3; three Level-1 options `InitWorkflow`, `AddTrigger`, `AddConditionalAction`):

```
import tracecat
wf = (tracecat
    .Workflow("ip_check")
    .add_trigger(
        tracecat.WebhookTrigger("/incoming"))
    .add_action(
        tracecat.ThreatIntelLookup(
            input="trigger.body.src_ip"))
    .add_action(
        tracecat.SlackNotify(
            channel="#alerts",
            condition=(
              "lookup.malicious == true")))
    .compile())
```

The Flat GRPO version reimplements the threat-intel lookup as a raw `requests.get` call, constructs trigger and action dicts via mutation, and never discovers the chained-builder idiom; the Option-GRPO version chains Tracecat's first-class actions through three discovered Level-1 options.

Aggregate slop reduction across the three examples. Token counts drop 71% (82→24), 67% (146→48), and 73% (287→78) respectively. Non-SDK calls drop from 5 to 0 across the three examples. Redundant intermediates drop from 13 total to 0 total. ODL drops from ∞ (no valid decomposition under the option set) to small finite values (1–3 options). These three examples are representative; Spearman correlations across all 5,042 passing solutions are reported in Figure 5.

D.5 Gradient Variance Analysis

Figure 4 plots the per-step relative gradient norm variance (computed as the ratio of the gradient-norm variance to the squared mean gradient norm, equivalent to the inverse SNR squared) for flat GRPO and Option-GRPO. Option-GRPO maintains lower variance throughout training, with the gap widening from 1.4× at initialization to 2.7× by step 20,000. This is consistent with the $\sqrt{T/K}$ improvement predicted by Proposition 3.

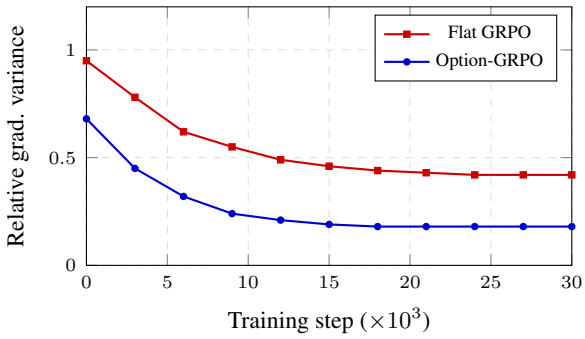

Figure 4: Per-step relative gradient norm variance (averaged over 12 SDKs). Option-GRPO shows roughly 2.7× lower variance at the end of training, consistent with the $\sqrt{T/K}$ SNR improvement of Proposition 3.

D.6 Full Ablation and Per-SDK Tables

Table 9: Hierarchy ablation. Both options and curriculum contribute; their combination yields the largest gains.

Config.	p@1	CL ↓	ID ↑	Conv. (steps)
Flat GRPO	52.7	71.3	0.41	24,800
Curric. only	55.4	66.8	0.49	19,200
Options only	58.1	58.7	0.66	15,600
Full (Option-GRPO)	**61.4**	**52.0**	**0.74**	**12,400**

Table 10: Cross-SDK option transfer (pass@1 %). Rows: source SDK; columns: target SDK.

Source → Target	Daft	LanceDB	cuallee
No transfer (from scratch)	69.5	72.6	75.4
Daft	–	74.1	76.2
LanceDB	71.2	–	76.8
cuallee	70.4	73.5	–

D.7 Empirical Validation of the Compression Results

Figure 5 plots $\text{ODL}_{\mathcal{H}}(c)$ against the four code quality metrics defined in Definition 4, computed over

Table 11: Comparison with prior RL methods on UnseenSDK-Bench. *Adapted to use our layered execution reward in place of unit tests. All use the same Qwen3-32B base.

Method	p@1	p@10	Critic	CL ↓	ID ↑
CodeRL-adapted*	46.8	63.2	Yes	79.5	0.38
PPOCoder-adapted*	48.5	64.7	Yes	76.1	0.40
StepCoder-adapted*	51.3	67.9	Yes	72.8	0.45
ExploraCoder	50.2	67.5	No	78.4	0.43
ReCode	55.1	71.2	No	74.1	0.51
Flat GRPO	52.7	68.4	No	71.3	0.41
Option-GRPO	**61.4**	**76.8**	**No**	**52.0**	**0.74**

Table 12: Comparison with frontier LLMs. All evaluated with the same RAG context (top-10 source-code chunks).

Method	p@1	p@10	Δ
Frozen Qwen3-32B	28.5	42.1	−32.9
GPT-4o	54.6	69.4	−6.8
Claude 3.5 Sonnet	57.2	71.8	−4.2
Gemini 2.0 Flash	51.8	67.1	−9.6
ReCode (32B)	55.1	71.2	−6.3
Option-GRPO (32B)	**61.4**	**76.8**	–

all 5,042 passing solutions produced by Option-GRPO across the 12 SDKs. Theorem 2 and Corollary 1 predict monotonic relationships for Q_{conc} and Q_{api}; Remark 5 treats Q_{red} and Q_{idiom} as empirical predictions. We observe Spearman rank correlations of $\rho_{\text{conc}} = -0.71$, $\rho_{\text{api}} = -0.83$, $\rho_{\text{red}} = -0.62$, and $\rho_{\text{idiom}} = -0.79$ (all $p < 10^{-50}$), confirming the predicted negative association between ODL and each quality metric in practice.

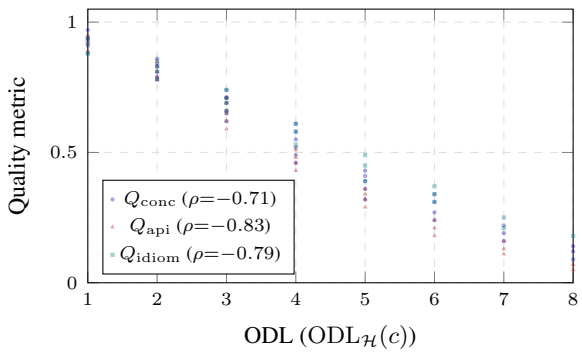

Figure 5: Code quality metrics against ODL on passing solutions. All four metrics show strong negative correlation with ODL, confirming Theorem 2 (conciseness), Corollary 1 (API efficiency), and the empirical predictions of Remark 5 (Q_{red} omitted from plot for legibility; $\rho_{\text{red}} = -0.62$, $p < 10^{-50}$).

E Detailed Artefact Release

We commit to releasing the following artefacts upon acceptance, under permissive licences, at a public GitHub repository linked from the camera-ready version (the main paper's brief summary appears just before the bibliography):

- **UnseenSDK-Bench**: 12 SDKs × three curriculum levels of tasks, totalling 1,871 Level-0 tasks, 3,742 Level-1 tasks, and 400 Level-2 tasks, with 7,484 auto-generated property tests, Docker sandbox configurations per SDK, and the layered execution reward implementation. Released under Apache 2.0.
- **Option discovery and extraction code** (Algorithms 1, 2, 3): parameter-flow graph construction, candidate mining, trajectory-based certification. Apache 2.0.
- **Training code**: Option-GRPO and all baselines (Flat GRPO, ReCode reimplementation, CodeRL/PPOCoder/StepCoder adaptations) on top of a publicly available GRPO framework, with full hyperparameter configurations (Table 3). Apache 2.0.
- **Evaluation harness** (benchmark task runner, property-test scoring, pass@k and code-quality metric computation) and the exact prompts used for the contamination probe (Appendix C.2). Apache 2.0.

Methodological details. All experiments report mean values across 3 random seeds per SDK; per-seed variance is below ±1.2 points pass@1 in every configuration we tested. Hardware is 8× NVIDIA H100 80GB with ZeRO-3 sharding and bf16 mixed precision; the full per-phase compute budget appears in Table 15, totalling roughly 7,632 H100-hours for the headline 12-SDK 32B run (or ∼ 5,500 H100-hours when cross-SDK transfer is exploited). Carbon estimate at 0.42 kgCO_2 per H100-hour: ∼ 3,205 kgCO_2e total. We adhered to each SDK's terms of use; none of the 12 SDKs are training-data-licensed, and all permit static analysis and benchmarking. No human subjects were involved in any phase of the work.

F Computational Details

F.1 Detailed Cost Breakdown

F.2 Scaling Analysis

Training cost scales with SDK size primarily through two factors. First, the number of Level 0

Table 13: Per-SDK results on UnseenSDK-Bench (pass@1 %). ExpC. = ExploraCoder.

SDK	Frozen	ExpC.	ReCode	Flat GRPO	Opt-GRPO	Δ vs ReCode	Category
cuallee	43.2	62.5	66.8	64.1	**75.4**	+8.6	Data Valid.
LanceDB	36.8	58.7	64.3	61.2	**72.6**	+8.3	Vector DB
Marker	41.5	59.8	65.1	62.7	**70.8**	+5.7	Doc Proc.
Surya	33.7	53.4	58.6	56.0	**63.2**	+4.6	Comp. Vision
Daft	30.6	54.8	61.4	58.2	**69.5**	+8.1	DataFrame
Vortex	27.4	48.9	54.2	51.6	**61.0**	+6.8	Scientific
libdebug	24.8	46.2	51.7	49.8	**58.4**	+6.7	Debugging
FastStream	22.1	43.7	48.5	46.9	**54.2**	+5.7	Messaging
Docling	18.5	40.6	46.8	43.4	**52.1**	+5.3	Doc Proc.
reflexio	16.2	38.4	43.5	40.7	**49.6**	+6.1	Agent Frmwk
Tracecat	11.7	34.1	40.2	37.4	**46.8**	+6.6	Security
MCP Python	9.4	31.6	40.1	40.4	**43.2**	+3.1	Protocol
Average	28.5	50.2	55.1	52.7	**61.4**	+6.3	–

Table 14: Complementarity with retrieval at inference time.

Configuration	p@1	p@10
RAG only (32B)	41.3	56.8
DocPrompting only (32B)	44.8	60.2
Option-GRPO (no retrieval)	61.4	76.8
Option-GRPO + RAG	**63.7**	**78.5**
Option-GRPO + DocPrompting	63.2	78.1

Table 15: Estimated per-phase computational cost.

Phase	Steps	Smp/Step	GPU-hrs
Graph construction	–	–	< 0.1
Option discovery	–	–	< 0.1
Property test gen.	–	–	< 0.1
Level 0 training	5,000	16	113.6
Level 1 training	10,000	16	198.4
Level 2 training	15,000	16	324.0
Total per SDK	30,000	–	636.0
Total all 12 SDKs	360,000	–	7,632

and Level 1 tasks grows with the number of public functions, increasing task distribution diversity but not the number of training steps (we train for a fixed step budget, sampling from the distribution). Second, execution cost per sample depends on SDK complexity: data processing SDKs (Daft, LanceDB) execute in under 1 s per sample, while SDKs with I/O operations (Docling, Marker) may require 5–10 s, impacting wall-clock time linearly.

The dominant cost is Level 2 training (15,000 steps $\times$ 16 samples), which requires generating and executing 240,000 complete programs per SDK. With an average execution time of 1.8 s per sample and 8-way GPU parallelism, the wall-clock time for Level 2 is approximately 40.5 hours per SDK.

Cross-SDK option transfer (Section D.6) can amortise costs: when source and target SDKs share a domain, training time on the target is reduced by approximately 28%, so the total cost for 12 SDKs falls from 7,632 to roughly 5,500 H100-hours when transfer is exploited across same-domain pairs.

F.3 Cost Sensitivity

The default training budget (30,000 total steps on a Qwen3-32B-Base, $\sim$ 80 H100-hours per SDK) is calibrated for the strongest pass@1. The method does not require this budget. Table 16 sweeps the training-step budget at fixed 32B scale; Table 17 sweeps the dense base-model size at fixed step budget. Both tables average across the 12 SDKs. All variants are dense Qwen3 models; we do not test MoE variants here to keep the size axis interpretable (parameters and active parameters coincide for dense models).

Table 16: Training-budget sweep on Qwen3-32B-Base (dense). Step counts are total across all three curriculum phases; we allocate $1/6$, $1/3$, $1/2$ of the budget to Phases $1, 2, 3$ respectively. The Flat GRPO baseline runs at the same total step budget and the same 32B base.

Total steps	GPU-hr	Flat	Opt	Δ
5,000	14	33.8	41.2	+7.4
10,000	28	44.6	54.1	+9.5
15,000	42	49.3	58.7	+9.4
20,000	55	51.2	60.4	+9.2
30,000 (default)	80	52.7	**61.4**	**+8.7**
45,000	122	53.1	62.1	+9.0

Three observations. (a) The hierarchical structure carries more weight than additional training steps. At 10,000 steps ($\sim$ 28 H100-hours), Option-GRPO reaches 54.1% pass@1, 1.4 points above Flat GRPO at full compute (52.7% at 30,000 steps). The smallest training budget at which Option-GRPO beats Flat-GRPO-at-default is

Table 17: Base-model-size sweep at default (30,000 step) budget. All variants are dense models from the Qwen3 family. GPU-memory footprint is per-replica peak during training with ZeRO-3 sharding on 8×H100.

Base	Params	Mem	Flat	Opt	Δ
Qwen3-8B	8.2B	20 GB	43.2	52.6	+9.4
Qwen3-14B	14.8B	36 GB	48.9	57.3	+8.4
Qwen3-32B	32.8B	75 GB	52.7	**61.4**	**+8.7**

roughly one-third the default compute. Practitioners with limited GPU access can therefore obtain Flat-GRPO-equivalent quality at a third of the cost, or better-than-Flat quality at full cost.

(b) The option vocabulary is the dominant gain, not scale. Option-GRPO on the 8B dense base (52.6%) is within 0.1 points of Flat GRPO on the 32B dense base (52.7%). The roughly 3.8× smaller GPU-memory footprint of the 8B variant makes Option-GRPO deployable on commodity hardware (a single 24GB GPU after 4-bit quantization) without sacrificing the absolute pass@1 level of the full-scale Flat baseline.

(c) Returns saturate beyond the default budget. Increasing from 30,000 to 45,000 total steps yields only +0.7 points pass@1 (61.4 → 62.1) at 1.5× the cost. The 30,000-step default sits at the elbow of the cost–performance curve. The Pareto frontier across budget and base-size is shown implicitly by the table: an 8B-Option run at 30,000 steps (52.6%, 20GB) dominates a 32B-Flat run at 30,000 steps (52.7%, 75GB) on the cost-adjusted criterion "pass@1 per GB-of-memory."

Implications for industry deployment. Combining (a) cross-SDK option transfer (Section D.6) and (b) the Qwen3-8B dense base brings the marginal cost of adapting Option-GRPO to a new SDK to ∼ 15 H100-hours plus ∼ 30 minutes of CPU graph construction: comparable to a single inference batch on a 32B frontier model, making per-SDK adaptation tractable for most engineering teams targeting a small set of high-traffic libraries.

Algorithm 4 Option-GRPO Training

Require: SDK source code $\mathcal{S}$, base model π_{base}, group size G, advancement threshold ρ_{adv}
Ensure: Trained hierarchical policy π_θ

// Stage 0: Graph Construction and Option Discovery
Construct G_{api}, G_{flow}, G_{sem} from $\mathcal{S}$
Discover Level 0 options $\mathcal{O}_0$ from public functions
Discover Level 1 candidates $\mathcal{O}_1^{\text{cand}}$ via Algorithm 1
Generate curriculum tasks $\mathcal{D}_0$, $\mathcal{D}_1$, $\mathcal{D}_2$
Generate property-based tests via Algorithm 3

// Phase 1: Atomic Option Learning (Level 0)
Initialize $\pi_\theta \leftarrow \pi_{\text{base}}$
for step $= 1$ **to** N_0 **do**
  Sample $x \sim \mathcal{D}_0$; generate G samples; execute; compute R_{exec}
  Compute GRPO advantages and update π_θ
end for
Extract $\mathcal{O}_0$ via Algorithm 2; partially freeze

// Phase 2: Pattern Learning (Level 1)
$\ell_{\text{curr}} \leftarrow 2$
for step $= 1$ **to** N_1 **do**
  Sample $x \sim \mathcal{D}_1$ with length $\leq \ell_{\text{curr}}$
  Generate G option sequences (action space $\mathcal{A}_1$)
  Expand, execute, compute $R_{\text{exec}} + \lambda_q R_{\text{quality}}$
  Compute sequence-level and option-level advantages
  Update π_θ via Equation 5
  if pass rate $> \rho_{\text{adv}}$ **then**
    $\ell_{\text{curr}} \leftarrow \min(\ell_{\text{curr}} + 1,\ L_{\max})$
  end if
end for
Extract $\mathcal{O}_1$ via Algorithm 2; partially freeze

// Phase 3: Program Learning (Level 2)
for step $= 1$ **to** N_2 **do**
  Sample $x \sim \mathcal{D}_2$
  Generate G option sequences (action space $\mathcal{A}_2$)
  Expand, execute, compute $R_{\text{exec}} + \lambda_q R_{\text{quality}} + \lambda_c R_{\text{consistency}}$
  Compute hierarchical advantages and update π_θ
end for
return π_θ