GRAPH PROGRAM SYNTHESIS: ADAPTIVE EXECUTION AND EVOLUTION FOR LONG-HORIZON MULTI-AGENT SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model-based multi-agent systems increasingly address long-horizon tasks, yet most deploy workflows that remain fixed after construction. This rigidity amplifies cascading failures, redundant information flow, and inefficient resource use as tasks or environments change. We introduce **Graph Program Synthesis (GPS)**, a program-centric framework that represents task solving as an evolving executable graph. GPS couples two capabilities: *Program Graph Representation*, which unifies computation units, dependencies, local state, tools, and model bindings, and *Reinforcement Learning-based Adaptive Program Evolution*, which formulates execution and semantic rewrites as a constrained sequential decision process. Transient executors perform bounded computation and commit validated artifacts, while a deterministic compiler activates candidate graphs only after structural, interface, state-transition, and budget checks. Across nine benchmarks spanning reasoning, code generation, multi-hop question answering, and tool use, GPS improves average task performance by 2.34 percentage points, reduces matched-performance cost by 34.8%, and raises final success under controlled runtime failures by 13.53 percentage points. These findings support a shift from static communication topologies toward adaptive executable programs for long-horizon multi-agent computation.

1 INTRODUCTION

Large language model-based multi-agent systems increasingly tackle long-horizon tasks in software development, scientific exploration, code generation, multi-hop question answering, and tool-augmented research. Such tasks require extended computation across heterogeneous models, external tools, intermediate artifacts, and dynamic environments. Systems coordinate specialized agents through role-based conversations and predefined workflows (Li et al., 2023; Chen et al., 2024; Wu et al., 2023; Hong et al., 2024; Qian et al., 2024). As these systems become computational infrastructures, however, the central challenge shifts from deciding *how agents communicate* to determining *how a complete task-solving process should be represented, executed, and reorganized over time*.

An agent-centric view is insufficient for this setting. Long-horizon execution requires task decomposition, intermediate artifacts, persistent state, model and tool allocation, and runtime adjustment. These decisions collectively define an executable program with explicit dependencies, state, and resource bindings. Unlike a transcript, this program must expose readiness, artifact ownership, and the conditions for downstream computation. The *task-solving program itself* should therefore be the primary object of organization, not merely the agents assigned to individual computations.

Recent work automates both agent programs and communication structures. ADAS searches over agentic programs, while GPTSwarm optimizes prompts and graph connectivity (Hu et al., 2024; Zhuge et al., 2024). AgentPrune removes redundant edges, G-Designer constructs task-dependent topologies, and Graph-of-Agents jointly selects models and directed message paths under task-specific objectives (Zhang et al., 2025b;c; Yun et al., 2026).

These methods establish that multi-agent organization is optimizable. Yet most optimize a candidate program or topology before deployment, emphasizing participation and information flow rather than the lifecycle of active computation. They specify which operations run, how outputs flow, and how

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

results are aggregated, but do not generally expose committed artifacts, readiness conditions, or validated runtime migration. Long-horizon organization therefore requires explicit execution state and controlled evolution of the active program.

Three limitations follow from this gap. Organizational rigidity arises because systems generate workflows before execution and keep them largely unchanged; they cannot reorganize computation when subproblems emerge, tools fail, or intermediate results prove invalid (Wang et al., 2026). Adaptation requires operations that add verification, split complex work, replace failed resources, and revise dependencies without discarding valid state.

Organizational information explosion arises when dense communication propagates redundant context and computation. The effectiveness of communication pruning (Zhang et al., 2025b) suggests that edges should reflect computational dependencies rather than unrestricted interaction. Decoupled structural and resource decisions arise when task decomposition is separated from model selection, tool assignment, scheduling, and context allocation. A unified mechanism coordinates program structure, execution state, and resources under quality, cost, latency, and reliability constraints.

These limitations motivate one question: *How can long-horizon multi-agent tasks be represented as executable programs and continuously reorganized during execution while preserving executability?* A solution requires a program-centric representation that exposes structure, state, artifacts, and resources; a sequential decision model because each action changes subsequent computation; and deterministic enforcement of dependency, interface, state-transition, and resource constraints. Because structural choices have delayed consequences, the adaptation policy must reason over quality, cost, latency, and failure risk throughout the remaining execution horizon.

We answer this question with Graph Program Synthesis (GPS), a framework that organizes long-horizon task solving as an evolving executable program. GPS has two pillars. *Pillar I.* Program Graph Representation represents task solving as a persistent executable graph. Nodes encode computation semantics, local state, authorized tools, and model configurations; directed edges specify execution dependencies rather than unrestricted communication. This representation preserves program-level state while supporting localized execution.

Pillar II. Adaptive Program Evolution formulates execution and restructuring as a constrained decision process for reinforcement learning. The evaluated prototype instantiates this formulation with a deterministic rather than learned controller. Semantic rewrites can add missing computation, decompose overloaded operations, merge redundant components, or revise failed dependencies. A deterministic runtime validates candidate versions before migration, keeping policy-driven adaptation within executable constraints throughout each proposed graph migration.

We evaluate GPS on nine benchmarks spanning reasoning, code generation, multi-hop question answering, and long-horizon tool use. GPS improves average task performance by 2.34 percentage points over the strongest multi-agent organization baseline and provides favorable performance–cost operating points. Under controlled runtime failures, it improves final task success by 13.53 percentage points over the strongest adaptive baseline. Ablation and transfer analyses further verify the contributions of program evolution, persistent state, and constraint-aware execution. These comparisons span task quality, resource regimes, controlled failures, component ablations, and transfer analyses across settings. The resulting evidence supports three contributions:

- **Program-centric formulation.** We formulate long-horizon multi-agent computation as a *versioned executable program graph* that unifies computation, dependencies, persistent state, artifacts, tools, and model bindings. This representation turns intermediate results into organizational memory while transient executors perform bounded computation.
- **Constraint-aware adaptive synthesis.** GPS enables the *joint adaptation of execution and graph topology* through a reinforcement learning policy. Its deterministic compiler validates semantic rewrites against structural, interface, state, and resource constraints, providing *compiler-validated program evolution* with state migration and rollback.
- **Comprehensive empirical validation.** Across nine benchmarks, GPS achieves a *2.34-point average performance improvement* with favorable performance–cost trade-offs and a *13.53-point success improvement under runtime faults*. Ablation and transfer studies further identify the contributions of graph evolution, persistent state, constraint-aware execution, and transferable organization policies

2 PRELIMINARIES

2.1 PROBLEM FORMULATION AND NOTATION

Let $q \in \mathcal{Q}$ denote a task for which the system produces a terminal output y and receives utility $R_{\text{task}}(y; q)$. We call q *long-horizon* when its solution comprises interdependent computations and future choices depend on artifacts, failures, verifier outcomes, or resource availability observed at runtime rather than being fixed before deployment.

At step t, the active organization is a directed attributed graph $\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t, \Phi_t)$: $\mathcal{V}_t$ contains computation units, $\mathcal{E}_t$ specifies data and control dependencies, and Φ_t collects their semantic configurations. Operational state X_t, recent feedback F_t, and remaining budget B_t define the observation $s_t = (\mathcal{G}_t, X_t, F_t, B_t)$. An action $a_t \in \mathcal{A}_{\text{exec}} \cup \mathcal{A}_{\text{evo}}$ advances the program or changes its organization, inducing $(\mathcal{G}_{t+1}, s_{t+1})$ and contributing to trajectory τ.

The objective is to learn or instantiate a policy π that maximizes terminal task quality while controlling cumulative cost $\mathcal{C}(\tau)$ and critical-path latency $\mathcal{L}(\tau)$:

$$\max_{\pi} \ \mathbb{E}_{\tau\sim\pi}[R_{\text{task}}(y; q) - \lambda_c \mathcal{C}(\tau) - \lambda_\ell \mathcal{L}(\tau)], \quad \begin{smallmatrix} \mathcal{G}_t \in \mathfrak{G}_{\text{valid}}, \\ a_t \in \Omega(s_t),\ \forall t, \end{smallmatrix} \tag{1}$$

where $\lambda_c, \lambda_\ell \geq 0$ select an operating point, $\mathfrak{G}_{\text{valid}}$ contains executable graphs, and $\Omega(s_t)$ enforces structural, lifecycle, and hard-budget constraints. Section 3 instantiates this formulation through persistent state, transient execution, semantic rewrites, and compiler-validated transitions.

2.2 RELATED WORK

Structured and automatically designed agent systems. Role-based systems coordinate specialized agents through conversations, standard operating procedures, debates, or staged development workflows (Li et al., 2023; Chen et al., 2024; Wu et al., 2023; Hong et al., 2024; Qian et al., 2024). ADAS, AFlow, MaAS, and FlowReasoner reduce manual design by searching agentic programs, optimizing code-represented workflows, or generating query-specific systems (Hu et al., 2024; Zhang et al., 2024; 2025a; Gao et al., 2025); MAS^2 additionally rectifies deployed systems from runtime feedback (Wang et al., 2026). GPS differs by making the active executable program, including committed artifacts and execution state, the persistent object of adaptation.

Graph-based multi-agent organization. Graph formulations expose participation and information flow as optimizable structure. GPTSwarm optimizes prompts and graph connectivity (Zhuge et al., 2024); AgentPrune removes redundant spatial–temporal communication (Zhang et al., 2025b); G-Designer generates task-conditioned topologies (Zhang et al., 2025c); and Graph-of-Agents selects models and constructs directed message paths (Yun et al., 2026). Their nodes primarily denote agents or model operations, and edges govern response exchange. GPS gives the graph executable semantics: nodes retain computation-local state, while edges determine readiness, artifact visibility, and downstream execution throughout the active program lifecycle.

Learning and feedback-driven adaptation. Learning-based organization optimizes graph connectivity, communication sparsity, topology generation, and query-level workflow construction (Zhuge et al., 2024; Zhang et al., 2025b;c; Gao et al., 2025). DyLAN selects agents and communication paths during inference, while MAS^2 rectifies systems from execution feedback (Liu et al., 2024; Wang et al., 2026). GPS instead applies semantic operations to a versioned executable program. Since unrestricted mutation can introduce cycles, unreachable computations, stale state, or budget violations, its policy proposes actions while a deterministic compiler admits only feasible program versions for subsequent execution within the running task.

3 METHODOLOGY

Overview. Graph Program Synthesis (GPS) treats long-horizon task solving as an executable program that can be revised, rather than a fixed agent team. Given a task q, GPS constructs $\mathcal{G}_0$, executes ready computations, observes their outcomes, and reorganizes the program when runtime evidence exposes an inadequate structure. Figure 1 summarizes this closed loop:

$$\begin{gathered} \mathcal{G}_0 \leftarrow \text{Init}(q), \qquad s_t \leftarrow \text{Observe}(\mathcal{G}_t), \qquad a_t \sim \widetilde{\pi}_\theta(\cdot \mid s_t), \\ \mathcal{G}_{t+1} \leftarrow \text{CommitIfValid}(\mathcal{G}_t, \text{Rewrite}(\mathcal{G}_t, a_t)) \,. \end{gathered} \tag{2}$$

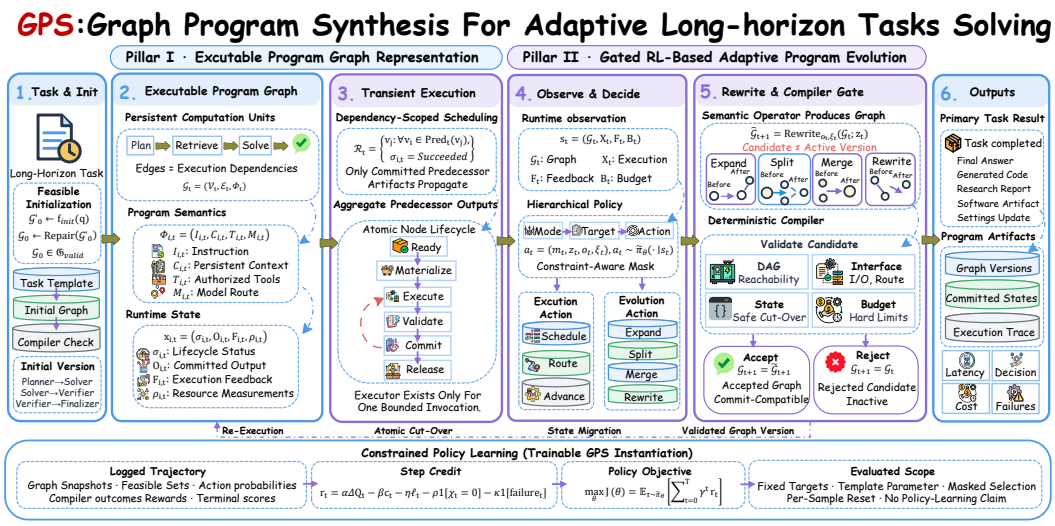


Figure 1: Overview of GPS. A persistent program graph organizes computation and artifacts; transient executors evaluate dependency-ready nodes; runtime feedback guides semantic graph revisions; and a deterministic compiler admits only valid versions before execution resumes.

The method has two complementary pillars. The first turns a task into a persistent executable graph whose dependencies make computation explicit. The second uses execution feedback to choose a small set of meaningful graph revisions, while a deterministic compiler prevents unsafe versions from entering execution. Together, they form an intuitive *represent–execute–observe–revise* cycle.

3.1 Pillar I: Executable Program Graph

Graph as the persistent program. At step t, GPS maintains a directed attributed graph $\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t, \Phi_t)$, following the formulation in Section 2.1. Each node $v_i \in \mathcal{V}_t$ represents a computation such as planning, solving, checking, or aggregation; each edge $e_{ij} \in \mathcal{E}_t$ means that node v_j depends on the committed result of v_i. A node separates what should be computed from what has happened during execution:

$$\underbrace{\Phi_{i,t}}_{\text{computation specification}} = (I_{i,t}, C_{i,t}, T_{i,t}, M_{i,t}), \qquad \underbrace{x_{i,t}}_{\text{execution record}} = (\sigma_{i,t}, O_{i,t}, F_{i,t}, \rho_{i,t}). \tag{3}$$

The specification contains the instruction $I_{i,t}$, local context $C_{i,t}$, authorized tools $T_{i,t}$, and model binding $M_{i,t}$. The execution record stores the node status $\sigma_{i,t}$, validated output $O_{i,t}$, feedback $F_{i,t}$, and resource use $\rho_{i,t}$. This separation is central to GPS: the graph may change, but verified intermediate work remains a reusable part of the program state instead of being lost in transient agent messages.

Edges define both order and information flow. Let $\mathrm{Pred}_t(v_j)$ denote the active predecessors of v_j. A node becomes ready only after all of its required predecessors have succeeded, and its input is assembled only from their committed artifacts:

$$\mathcal{R}_t = \{v_j \in \mathcal{V}_t : \forall v_i \in \mathrm{Pred}_t(v_j),\ \sigma_{i,t} = \mathrm{Succeeded}\}, \tag{4}$$

$$Z_{j,t} = \mathrm{Aggregate}(\{O_{i,t} : v_i \in \mathrm{Pred}_t(v_j)\}) \otimes \Phi_{j,t}, \qquad v_j \in \mathcal{R}_t. \tag{5}$$

Here, $\otimes$ binds the available artifacts to the node specification. Consequently, the same edge has a clear operational meaning: it controls when a computation can run and which information it can receive. This avoids unrestricted broadcast, reduces irrelevant context propagation, and makes the execution trace directly readable from the graph.

Persistent artifacts, transient execution. The graph is persistent, whereas an executor exists only for one bounded node invocation. For each ready node, GPS materializes the required prompt and

resources, executes the computation, and commits the result only if it satisfies the node contract:

$$\epsilon_{j,t} = \text{Materialize}(\Phi_{j,t}, Z_{j,t}), \qquad (\widehat{O}_{j,t}, F_{j,t}) = \text{Execute}(\epsilon_{j,t}),$$
$$x^{+}_{j,t} = \begin{cases} \text{Commit}(\widehat{O}_{j,t}, F_{j,t}), & \text{Validate}(\widehat{O}_{j,t}) = 1, \\ \text{RecordFail}(F_{j,t}), & \text{otherwise.} \end{cases} \tag{6}$$

The executor is then released. Successful outputs become durable graph artifacts available to downstream nodes and later graph versions; failed or contract-violating outputs are recorded as feedback but cannot contaminate subsequent computation. The representation supports tool permissions through $T_{j,t}$, although the evaluated prototype does not optimize tool routing.

Feasible initialization. GPS begins from a compact task-conditioned template rather than attempting to discover the entire organization before execution. The initializer proposes a planner–solver–verifier–finalizer graph appropriate for the task family, and the same compiler used during evolution checks it before activation:

$$\widehat{\mathcal{G}}_0 = f_{\text{init}}(q), \qquad \mathcal{G}_0 = \widehat{\mathcal{G}}_0 \quad \text{only if} \quad \text{Valid}(\widehat{\mathcal{G}}_0) = 1. \tag{7}$$

An invalid proposal is rejected rather than executed. Initialization therefore provides a legal starting structure, while task-specific organization is refined from evidence collected during the actual run.

3.2 Pillar II: Reinforcement Learning-based Program Evolution

Feedback-driven decisions. When execution succeeds, GPS simply advances through the ready set. A failure, verifier rejection, or output-contract violation creates an evolution decision point. The policy observes the current graph together with node outcomes, resource pressure, and remaining budget,

$$s_t = \big(\mathcal{G}_t, X_t, F_t, B_t\big), \qquad a_t = (z_t, o_t, \xi_t) \sim \widetilde{\pi}_\theta(\cdot \mid s_t), \tag{8}$$

where z_t is the affected graph location, o_t is a semantic rewrite, and ξ_t contains its local parameters. This formulation keeps the decision interpretable: the controller identifies *where* the program is ineffective and chooses *how* to reorganize it. The core operators are Expand, which adds missing computation; Split, which decomposes an overloaded node; Rewire, which replaces an ineffective dependency; and NoOp, which preserves the current structure when intervention is unnecessary.

Only feasible revisions are considered. The legal action set $\Omega(s_t)$ is determined by the current topology, node lifecycle, experimental setting, and resource limits. Rather than asking the policy to learn hard safety rules through failure, GPS masks illegal actions before selection:

$$\widetilde{\pi}_\theta(a \mid s_t) = \frac{\pi_\theta(a \mid s_t)\, \mathbf{1}[a \in \Omega(s_t)]}{\sum_{a' \in \mathcal{A}} \pi_\theta(a' \mid s_t)\, \mathbf{1}[a' \in \Omega(s_t)]}, \qquad \text{NoOp} \in \Omega(s_t). \tag{9}$$

Masking narrows the policy to meaningful choices without replacing final validation. It also cleanly separates adaptive decisions, which benefit from experience, from invariants that must hold for every executable graph.

Compiler-gated graph evolution. The selected operator produces a candidate version rather than modifying the active graph in place. A deterministic compiler checks that the candidate remains acyclic, connected from entry to output, executable at every active node, and compatible with committed state; a runtime guard separately enforces the resource contract. The complete transition is

$$\widehat{\mathcal{G}}_{t+1} = \text{Rewrite}_{o_t, \xi_t}(\mathcal{G}_t; z_t), \tag{10}$$
$$\mathcal{G}_{t+1} = \begin{cases} \text{Migrate}(\mathcal{G}_t, \widehat{\mathcal{G}}_{t+1}), & \text{Valid}(\widehat{\mathcal{G}}_{t+1}) = 1, \\ \mathcal{G}_t, & \text{otherwise.} \end{cases} \tag{11}$$

An accepted candidate becomes the next graph version and inherits all compatible committed artifacts. A rejected candidate leaves the active program unchanged and supplies rejection feedback to the controller. Execution then resumes from the ready set. This proposal–validation–commit protocol allows GPS to reorganize computation at runtime without sacrificing deterministic safety or previously verified work.

Table 1: Main results on nine benchmarks. Values are official task metrics (%); higher is better. Within the shared-backbone block, arrows show the change from Direct Prompting (Vanilla), and bold/underlined values denote the best/second-best result. Cross-backbone rows are separated and report absolute scores only because a DeepSeek Vanilla delta would confound backbone and organization effects.

Method	Math				Code		Multi-domain	Multi-hop	Research
	GSM8K	SVAMP	AQuA	MATH	HumanEval	MBPP	MMLU	HotpotQA	BrowseComp+
Direct Prompting (Vanilla)	95.84	94.71	84.62	62.84	86.74	73.28	84.37	69.48	13.24
CoT (Zero-shot)	96.73 ↑0.89	95.48 ↑0.77	85.76 ↑1.14	65.37 ↑2.53	88.16 ↑1.42	75.43 ↑2.15	85.76 ↑1.39	70.92 ↑1.44	15.87 ↑2.63
Debate (3 agents)	96.28 ↑0.44	96.63 ↑1.92	86.41 ↑1.79	67.26 ↑4.42	89.37 ↑2.63	76.82 ↑3.54	86.31 ↑1.94	74.18 ↑4.70	14.76 ↑1.52
Self-Consistency	97.42 ↑1.58	97.11 ↑2.40	87.38 ↑2.76	68.44 ↑5.60	91.08 ↑4.34	78.64 ↑5.36	87.16 ↑2.79	73.46 ↑3.98	12.31 ↓0.93
AgentPrune	96.11 ↑0.27	95.32 ↑0.61	87.64 ↑3.02	64.88 ↑2.04	89.62 ↑2.88	76.31 ↑3.03	88.79 ↑4.42	72.36 ↑2.88	12.84 ↓0.40
GPTSwarm	96.84 ↑1.00	96.37 ↑1.66	87.13 ↑2.51	66.51 ↑3.67	91.47 ↑4.73	78.13 ↑4.85	88.07 ↑3.70	76.18 ↑6.70	15.62 ↑2.38
G-Designer	97.18 ↑1.34	96.05 ↑1.34	88.21 ↑3.59	69.37 ↑6.53	90.86 ↑4.12	79.42 ↑6.14	89.08 ↑4.71	77.54 ↑8.06	14.93 ↑1.69
GoA (Top-$k = 3$)	<u>97.61</u> ↑1.77	96.92 ↑2.21	<u>88.74</u> ↑4.12	<u>72.86</u> ↑10.02	92.31 ↑5.57	80.27 ↑6.99	<u>89.74</u> ↑5.37	80.72 ↑11.24	16.41 ↑3.17
MAS[2]	97.07 ↑1.23	<u>97.44</u> ↑2.73	88.36 ↑3.74	71.46 ↑8.62	<u>96.58</u> ↑9.84	<u>84.73</u> ↑11.45	89.51 ↑5.14	<u>89.08</u> ↑19.60	<u>19.43</u> ↑6.19
GPS (DeepSeek-v4-pro)	**97.94** ↑2.10	**97.83** ↑3.12	**90.42** ↑5.80	**74.92** ↑12.08	**97.36** ↑10.62	**86.92** ↑13.64	**90.86** ↑6.49	**92.11** ↑22.63	**26.37** ↑13.13
GPS (GPT-5.5)	**99.03**	**98.86**	93.74	**82.17**	**98.78**	**91.26**	**94.58**	**96.11**	**36.82**
GPS (Qwen3-14B)	96.32	95.63	85.18	66.51	82.67	70.84	82.46	68.24	17.18
GPS (DeepSeek + Qwen)	98.41	98.32	**93.86**	77.36	98.14	88.47	92.07	94.28	30.14

Learning objective. The step signal assigns credit to quality change while charging the resource and reliability consequences of an intervention:

$$r_t = \alpha \Delta Q_t - \beta c_t - \eta \ell_t - \rho \mathbf{1}[\chi_t = 0] - \kappa \mathbf{1}[\text{failure}_t], \qquad \max_\theta \; J(\theta) = \mathbb{E}_{\tau \sim \tilde{\pi}_\theta}\left[\sum_{t=0}^{T} \gamma^t r_t\right]. \quad (12)$$

ΔQ_t is used only when a calibrated verifier provides an intermediate quality signal; otherwise the terminal task reward supplies outcome supervision. Hard constraints remain in Ω and the compiler gate rather than becoming soft penalties that the policy may trade away. This objective supports policy-gradient or actor–critic training without tying GPS to a particular optimizer.

4 Experiments

We organize the evaluation around five questions. **RQ1** measures overall effectiveness across heterogeneous task families; **RQ2** examines the performance–cost frontier; **RQ3** tests runtime adaptation under controlled node faults; **RQ4** isolates the contributions of graph evolution, rewiring, and persistent state; and **RQ5** evaluates few-shot transfer of the organization policy to a new backbone. Together, these questions separate improvements in task quality from effects attributable to resource use, runtime conditions, architectural components, or backbone choice. This design examines when GPS gains arise, which components enable them, and whether they transfer across settings.

4.1 Experimental Setup

Benchmarks and metrics. We evaluate GPS on nine benchmarks spanning mathematical reasoning (GSM8K, SVAMP, AQuA, and MATH), code generation (HumanEval and MBPP), multi-domain reasoning (MMLU), multi-hop question answering (HotpotQA), and long-horizon research (BrowseComp+). We use each benchmark's official percentage-scale metric: accuracy for reasoning, Pass@1 for code generation, and the official task score for HotpotQA and BrowseComp+. RQ2, RQ4, and RQ5 use *Common-5*, the unweighted macro-average over GSM8K, MATH, HumanEval, HotpotQA, and BrowseComp+.

Baselines and controls. RQ1 compares GPS with Direct Prompting, zero-shot Chain-of-Thought (CoT), Debate, Self-Consistency, AgentPrune, GPTSwarm, G-Designer, Graph-of-Agents (GoA), and MAS^2. The primary comparison fixes the `DeepSeek-v4-pro` backbone, task instances, decoding configuration, and evaluation scripts; cross-backbone GPS variants are reported in a separate table block and are not interpreted as controlled organization-method comparisons. RQ2 evaluates representative low-, medium-, and high-resource operating points, RQ3 applies the same fault schedule to all systems, and RQ4 changes one architectural capability at a time. Implementation, accounting, and protocol details are deferred to Appendix D.

4.2 RQ1: Overall Effectiveness

GPS is the strongest method in every shared-backbone column. Its advantage over the best non-GPS baseline is modest on GSM8K and SVAMP (0.33 and 0.39 points), but grows on MATH (2.06), MBPP (2.19), HotpotQA (3.03), and BrowseComp+ (6.94). Figure 2 makes this effect-size profile explicit: the largest gains occur on the multi-hop and deep-research tasks, where execution dependencies are longer and failures propagate across more steps. The comparison therefore supports the value of executable program organization beyond model scaling, while the separate cross-backbone block shows that stronger or heterogeneous backbones can further raise the ceiling.

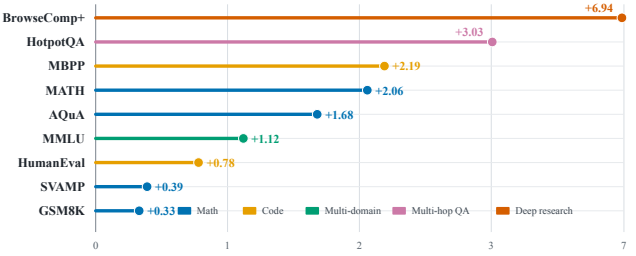


Figure 2: GPS gain over the strongest non-GPS baseline for each benchmark.

4.3 RQ2: Cost–Performance Trade-off

RQ2 evaluates all configurations on the same Common-5 suite: GSM8K, MATH, HumanEval, HotpotQA, and BrowseComp+. We report on their unweighted macro-average together with the total monetary cost and token consumption. The current evaluation compares representative operating points rather than a complete matched-budget grid; GPS is instantiated in *Lite*, *Balanced*, and *Full* resource regimes. As shown in Figure 3, Direct Prompting is the cheapest operating point but achieves only 65.63%. And GPS (Lite) reaches 70.41% at a cost of \$0.220, outperforming AgentPrune by 3.25 points for only \$0.002 additional cost. It also nearly matches GPTSwarm (70.41% versus 70.52%) while using 42.7% fewer tokens and 41.5% less monetary cost. GPS (Balanced) obtains 73.56% at \$0.392, exceeding GoA by 1.18 points while costing \$0.004 less.

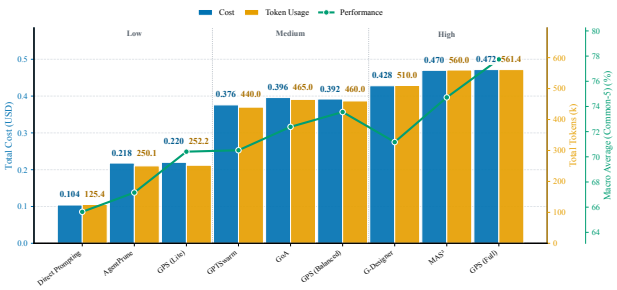


Figure 3: Cost-Performance Trade-off across Agent-Organization Methods on Common-5 Benchmarks.

GPS (Full) achieves the highest Common-5 score of 77.74%, exceeding MAS^2 by 3.02 points while requiring only \$0.002 additional cost and 1.4k additional tokens. Across its three configurations, GPS improves from 70.41% to 73.56% and then to 77.74%, showing that additional resources are converted into progressively higher task performance. GPS is not always the lowest-cost method. Instead, it provides favorable operating points across lightweight, balanced, and high-performance regimes. The results indicate that dependency-aware execution and adaptive program organization use additional computation more effectively than merely increasing communication or agent count.

4.4 RQ3: Runtime Adaptivity under Synthetic Node Faults

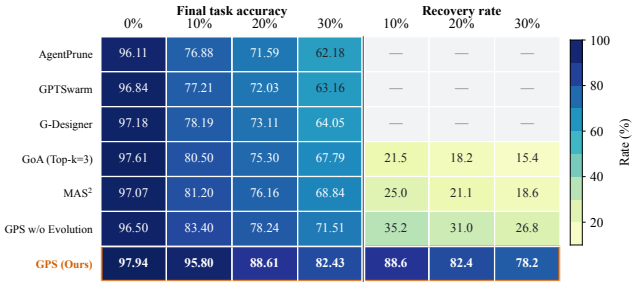


Figure 4: Runtime robustness and fault recovery under increasing per-node fault severity on GSM8K.

RQ3 independently injects synthetic failures into each eligible GSM8K execution node at per-node probabilities of 0%, 10%, 20%, and 30%. A triggered node cannot commit a valid output. Because one task may traverse multiple eligible nodes, the proportion of affected task instances can exceed the nominal injection probability. Figure 4 reports final task accuracy for all systems and recovery rate only for methods with an evaluated runtime intervention mechanism. The two column groups distinguish end-to-end robustness from post-failure recovery, while blank cells denote unavailable recovery measurements rather than zero recovery. Each column corresponds exactly to one measured fault setting, without interpolation between observations.

Clean performance is comparable: GPS attains 97.94%, versus 97.61% for GoA, 97.18% for G-Designer, and 97.07% for MAS^2. Under 30% faults, AgentPrune, GPTSwarm, and G-Designer fall to 62.18%, 63.16%, and 64.05%; GoA and MAS^2 reach 67.79% and 68.84%. GPS without evolution retains retry and state reuse but reaches only 71.51%, whereas full GPS maintains 82.43%. Across 10%, 20%, and 30% faults, full GPS exceeds the no-evolution variant by 12.40, 10.37, and 10.92 points and recovers 88.6%, 82.4%, and 78.2% of affected executions. Runtime graph evolution therefore provides the mechanism that remains effective as faults accumulate.

4.5 RQ4: Component Analysis

RQ4 evaluates controlled ablations on Common-5 with the same backbone, initialization, instances, and scoring protocol. Table 4.5 reports each absolute score together with its decrease from full GPS, using the same reference-delta convention as RQ1. Removing graph evolution produces the largest drop, from 77.74% to 69.78% (-7.96 points), nearly collapsing GPS to the static multi-agent graph at 69.98%. Removing rewiring or persistent state yields smaller decreases of 2.85 and 2.41 points, while removing both lowers performance by 5.07 points. Thus, graph evolution supplies the dominant gain, and rewiring and persistent state contribute complementary improvements rather than duplicating the same function.

Table 2: Component analysis of GPS on Common-5. The drop is measured relative to full GPS.

Architecture Configuration	Macro Avg.	Drop
Full GPS	**77.74**	–
w/o Dynamic Rewiring	74.89	2.85
w/o Graph Evolution	69.78	**7.96**
w/o Persistent Program State	75.33	2.41
w/o Persistent State and Rewiring	72.67	5.07
Static Multi-Agent Graph	69.98	7.76

4.6 RQ5: Few-shot Cross-Backbone Adaptation

RQ5 transfers the GPS organization policy from `DeepSeek-v4-pro` to Qwen3-14B on Common-5. The backbone, executor, compiler, prompts, and graph representation remain fixed during policy-only adaptation; only the execution and graph-evolution policy is updated from 16 or 32 target-backbone examples. Zero-shot transfer attains 58.20%, leaving a 7.98-point oracle gap. Policy-only adaptation reaches 62.80% with 16 examples and 64.85% with 32, recovering 83.3% of the initial gap. At 32 shots, full adaptation improves by only one additional point (65.85% versus 64.85%) while increasing adaptation cost from \$0.12 to \$0.28. The organization policy therefore captures most of the transferable benefit with limited target-backbone supervision and substantially lower adaptation cost than full adaptation. Increasing the policy-only

Table 3: Few-shot cross-backbone adaptation of GPS from DeepSeek-v4-pro to Qwen3-14B on Common-5.

Setting	Shots	Macro Avg.	Cost	Gap
Zero-shot Transfer	0	58.20	0.00	7.98
Policy-only Adapt.	16	62.80	0.06	3.38
Policy-only Adapt.	32	64.85	0.12	1.33
Full Adaptation	32	65.85	0.28	0.33
Target Oracle	Full	**66.18**	1.15	0.00

adaptation set from 16 to 32 examples yields a further 2.05-point gain and reduces the remaining oracle gap from 3.38 to 1.33 points. Notably, the 32-shot policy-only setting recovers most of the oracle performance at less than half the cost of full adaptation, highlighting the efficiency of adapting organizational decisions without modifying the remaining GPS components.

4.7 Case Study: Program Graph Evolution

To better understand how GPS performs runtime adaptation, we present a representative execution trajectory from a long-horizon reasoning task. As illustrated in Figure 5, GPS transforms runtime feedback into validated program-graph evolution rather than restarting the original execution path. Initially, GPS constructs a feasible program graph containing coarse computation units for planning, retrieval, reasoning, and verification. During execution, the runtime feedback indicates that the retrieved evidence is insufficient for reliable downstream reasoning. Instead of restarting the workflow or relying on repeated communication, GPS identifies a missing-computation defect and applies the Expand operator to insert an additional verification step. After compiler validation, the updated graph is activated while preserving previously committed intermediate artifacts through state migration. Only the affected downstream computations are re-executed. This example illustrates the key difference between GPS and static agent organizations: GPS does not only optimize who communicates with whom, but continuously revises the executable computation itself according to runtime evidence.

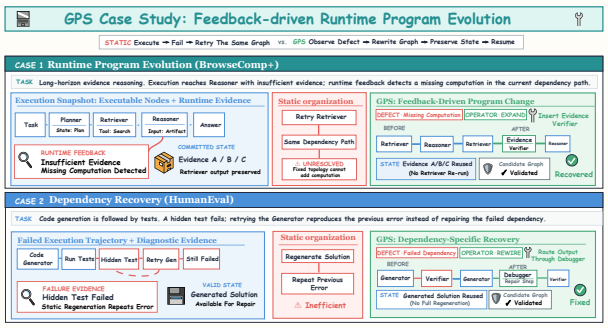


Figure 5: Runtime program evolution in GPS. Runtime feedback localizes program defects; GPS selects semantic rewrite operators, preserves valid intermediate state, validates the candidate graph, and resumes execution with an adapted executable program.

5 Conclusion

In this work, we introduced Graph Program Synthesis (GPS), a framework that formulates long-horizon multi-agent task solving as the execution and optimization of an evolving program graph. Instead of treating agents and communication links as a fixed organization, GPS represents task computation through explicit executable nodes, dependencies, and persistent program state, and uses a feedback-driven policy to jointly decide when to continue execution and when to revise the active program structure. Semantic rewrite operators and runtime validation further ensure that structural adaptations remain executable and consistent with previously committed state.

Across heterogeneous reasoning, code-generation, multi-hop question answering, and long-horizon research benchmarks, GPS consistently outperforms representative single-agent, multi-agent, and graph-organization baselines. The results further show that GPS provides favorable performance–cost operating points across different resource regimes, preserves substantially higher task accuracy under synthetic runtime failures, and benefits primarily from graph evolution, dynamic rewiring, and persistent program state. Moreover, the learned organization policy can be transferred to a different backbone with only a small number of target-backbone examples, indicating that the acquired program-organization behavior is not entirely tied to a single underlying model.

These findings position GPS as a general organization layer rather than a task-specific workflow. Future work can retain and optimize the policy across trajectories, extend semantic rewrites to richer tool-using programs, and study how program evolution transfers across domains, model families, and longer execution horizons under real operational failure settings. Taken together, these findings suggest a shift from optimizing static agent teams or communication topologies toward synthesizing executable multi-agent programs that can inspect their own execution, preserve useful intermediate computation, and revise their structure in response to runtime evidence.

REPRODUCIBILITY STATEMENT

We provide the implementation and evaluation details of Graph Program Synthesis (GPS) in the main text and appendix. All experiments use nine publicly available benchmarks: GSM8K, SVAMP, AQuA, MATH, HumanEval, MBPP, MMLU, HotpotQA, and BrowseComp+. Code, configurations, and evaluation scripts will be released to support reproduction.

AI USE STATEMENT

Generative AI tools were not used in the preparation of this manuscript or in the development of the method. GPS itself is a large language model-based multi-agent framework and therefore uses generative models as experimental backbones and transient executors. This use is an integral part of the evaluated methodology rather than assistance in manuscript writing or technical development.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021. URL https://arxiv.org/abs/2108.07732.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. URL https://arxiv.org/abs/2107.03374.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=EHg5GDnyq1.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, Sahel Sharifymoghaddam, Yanxi Li, Haoran Hong, Xinyu Shi, Xuye Liu, Nandan Thakur, Crystina Zhang, Luyu Gao, Wenhu Chen, and Jimmy Lin. BrowseComp-Plus: A more fair and transparent evaluation benchmark of deep-research agent. *arXiv preprint arXiv:2508.06600*, 2025. URL https://arxiv.org/abs/2508.06600.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021. URL https://arxiv.org/abs/2110.14168.

Hongcheng Gao, Yue Liu, Yufei He, Longxu Dou, Chao Du, Zhijie Deng, Bryan Hooi, Min Lin, and Tianyu Pang. FlowReasoner: Reinforcing query-level meta-agents. *arXiv preprint arXiv:2504.15257*, 2025.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *International Conference on Learning Representations*, 2021a. URL https://openreview.net/forum?id=d7KBjmI3GmQ.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021b. URL https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/be83ab3ecd0db773eb2dc1b0a17836a1-Abstract-round2.html.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, 2024.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative agents for "mind" exploration of large language model society. In *Advances in Neural Information Processing Systems*, volume 36, pp. 51991–52008, 2023. doi: 10.52202/075280-2264. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/a3621ee907def47c1b952ade25c67698-Abstract-Conference.html.

Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. Program induction by rationale generation: Learning to solve and explain algebraic word problems. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 158–167, Vancouver, Canada, 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1015. URL https://aclanthology.org/P17-1015/.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. A dynamic llm-powered agent network for task-oriented agent collaboration. *arXiv preprint arXiv:2310.02170*, 2024.

Arkil Patel, Satwik Bhattamishra, and Navin Goyal. Are NLP models really able to solve simple math word problems? In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2080–2094, Online, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.168. URL https://aclanthology.org/2021.naacl-main.168/.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.810. URL https://aclanthology.org/2024.acl-long.810/.

Kun Wang et al. Mas^2: Self-generative, self-configuring, self-rectifying multi-agent systems. In *International Conference on Learning Representations*, 2026.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, Brussels, Belgium, 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

Sukwon Yun et al. Graph-of-agents: A graph-based framework for multi-agent llm collaboration. In *International Conference on Learning Representations*, 2026.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025a.

Guibin Zhang et al. Cut the crap: An economical communication pipeline for llm-based multi-agent systems. In *International Conference on Learning Representations*, 2025b.

Guibin Zhang et al. G-designer: Architecting multi-agent communication topologies via graph neural networks. In *International Conference on Machine Learning*, 2025c.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*, 2024.

Mingchen Zhuge et al. Gptswarm: Language agents as optimizable graphs. In *International Conference on Machine Learning*, 2024.

A Appendix Roadmap and Notation

This appendix expands the compact presentation in Sections 2.1-4. Appendix B specifies graph construction and transient execution; Appendix C formalizes rewrite operators, RL training, and compiler-gated deployment; Appendix D documents benchmarks, baselines, metrics, and the protocols behind RQ1-RQ5.

Table 4: Core notation used by GPS. Graph semantics are persistent across a program version, whereas runtime state changes as nodes execute, fail, retire, or commit artifacts.

Symbol	Meaning
q, y	Input task and terminal system output.
$\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t, \Phi_t)$	Active directed program graph, execution dependencies, and node semantics at decision step t.
$\Phi_{i,t} = (I_{i,t}, C_{i,t}, T_{i,t}, M_{i,t})$	Instruction, persistent local context, authorized-tool field, and model route of node v_i.
$x_{i,t} = (\sigma_{i,t}, O_{i,t}, F_{i,t}, \rho_{i,t})$	Node lifecycle status, committed output, feedback, and resource measurements.
X_t, F_t, B_t	Program-wide operational state, recent feedback, and remaining hard budgets.
$\mathcal{R}_t$	Set of pending nodes whose active predecessors have succeeded.
$a_t = (m_t, z_t, o_t, \xi_t)$	Hierarchical decision: mode, target, semantic operator, and operator parameters.
$\Omega(s_t)$	State-dependent feasible action set after local preconditions and ablation masks.
$\widehat{\mathcal{G}}_{t+1}$	Candidate graph produced before compiler validation and cut-over.
χ_t	Boolean compiler-and-runtime acceptance indicator for a candidate graph version.

B Extended Specification of Graph Program Synthesis

B.1 Feasible Initial Program Construction

GPS begins from an executable program rather than an unconstrained communication graph. For a task q with task type $\kappa(q)$ and model route M, the initializer instantiates a task-conditioned planner–solver–verifier–finalizer template,

$$\widehat{\mathcal{G}}_0 = f_{\text{init}}\big(q, \kappa(q), M, \zeta\big), \qquad \zeta \in \{\textsc{Chain}, \textsc{Tree}, \textsc{Complete}\}, \tag{13}$$

where ζ selects the initial dependency pattern. The default chain uses one solver; the tree template exposes two solver branches to a shared verifier; and the complete template adds direct dependencies among the planner, solver, verifier, and finalizer while preserving a single entry and output. These templates establish a legal starting interface that the RL policy subsequently evolves from runtime feedback.

Table 5: GPS task contracts. The contract is propagated through solver, verifier, recovery, and finalizer nodes so that a rewrite cannot relax the required output type.

Task type	Expected deliverable	Validation focus
Numeric	One terminal numerical answer	Parseability and preservation of the declared numeric answer format.
Multiple choice	One uppercase option label	Membership in the available option set and single-label formatting.
Text QA	One concise answer string	Presence of a nonempty answer consistent with the benchmark scorer interface.
Code	One Python code block or requested continuation	Syntactic extraction followed by execution against the benchmark tests.

The compiler evaluates the proposed graph from its structural and interface state. Initialization succeeds only if

$$\mathrm{Compile}(\widehat{\mathcal{G}}_0) = \mathrm{accept}, \qquad \mathcal{G}_0 = \widehat{\mathcal{G}}_0 \in \mathfrak{G}_{\mathrm{valid}}. \tag{14}$$

An invalid template is rejected, and every graph change that affects execution is recorded in either the initial template or the explicit rewrite history.

Task and output contracts. The implementation maps each benchmark item to one of four output contracts. The contract controls final normalization and determines when an otherwise fluent response is treated as an execution failure. Table 5 summarizes this interface.

B.2 Persistent State and Dependency-Scoped Materialization

For an active node v_j, let $\mathrm{Pred}_t(v_j)$ contain only non-retired predecessors. Readiness is defined by committed predecessor state rather than by message arrival:

$$\begin{aligned}\mathcal{R}_t = \{v_j : \sigma_{j,t} = \mathrm{Pending},\\ \forall v_i \in \mathrm{Pred}_t(v_j),\ \sigma_{i,t} = \mathrm{Succeeded}\}.\end{aligned} \tag{15}$$

For $v_j \in \mathcal{R}_t$, GPS aggregates only predecessor artifacts visible along active incoming edges,

$$\begin{aligned}Z_{j,t} &= \mathrm{Serialize}\left(q, C_{j,t}, \{(i, O_{i,t}) : v_i \in \mathrm{Pred}_t(v_j)\}\right),\\ \epsilon_{j,t} &= \mathrm{Materialize}(I_{j,t}, M_{j,t}, T_{j,t}, Z_{j,t}).\end{aligned} \tag{16}$$

The graph remains the durable object; $\epsilon_{j,t}$ exists only for one bounded invocation. This design prevents an executor's undeclared conversational history from becoming hidden program state and makes the information visible to each computation recoverable from the graph version and its committed artifacts.

Atomic lifecycle. An invocation first moves the node from Pending to Running. The produced candidate artifact $\widehat{O}_{j,t}$ is committed only after provider, output-contract, and verifier checks succeed:

$$(\sigma_{j,t}, O_{j,t}, F_{j,t})^{+} = \begin{cases}(\mathrm{Succeeded}, \widehat{O}_{j,t}, F_{j,t}), & \mathrm{Valid}(\widehat{O}_{j,t}) = 1,\\ (\mathrm{Failed}, , F_{j,t}), & \mathrm{Valid}(\widehat{O}_{j,t}) = 0.\end{cases} \tag{17}$$

Thus, downstream nodes never consume an uncommitted candidate. A successful predecessor artifact persists across later retries and compatible graph versions; a failed node records feedback but contributes no output. Retired nodes remain in the version history while being excluded from active readiness and root-to-output checks.

Feedback taxonomy. GPS distinguishes adaptive failures from provider or configuration failures. Injected node failures, verifier rejection, and final-output contract violations create evolution decision points. Authentication, provider availability, and invalid configuration errors terminate the affected run, restricting RL adaptation to events inside the program's causal scope.

B.3 End-to-End Execution–Evolution Procedure

The runtime follows one state machine for clean execution and recovery:

1. construct and compile $\mathcal{G}_0$;
2. compute $\mathcal{R}_t$ and select the next ready computation under the declared scheduling rule;
3. materialize a transient executor from node semantics and committed predecessor artifacts;
4. execute within call, token, cost, retry, and rewrite caps;
5. atomically commit a valid artifact or record typed failure feedback;
6. on adaptive failure, form s_t, mask illegal actions, and select an evolution operator with the trained policy;
7. apply the operator to a copy of the active graph, compile the candidate, and either commit the new version or preserve the old one;
8. resume dependency-driven execution until the output node commits or a hard stop condition is reached.

Let $Y_t = (\mathcal{G}_t, X_t)$. The procedure can be summarized as the partial transition

$$Y_{t+1} = \begin{cases} \mathrm{ExecStep}(Y_t), & \text{node commit,} \\ \mathrm{CutOver}(\widehat{\mathcal{G}}_{t+1}, X_t), & \text{rewrite,} \\ Y_t, & \text{decline/reject,} \end{cases} \tag{18}$$

with termination determined independently by successful output, provider/configuration failure, or an exhausted hard budget.

C Safe Program Evolution

C.1 Hierarchical RL Decisions and Feasible-Action Masking

The general GPS policy factors a decision into execution-or-evolution mode, target, semantic operator, and operator parameters. During training, actions are sampled from the masked policy to optimize the return; during evaluation, the learned parameters are frozen and the highest-probability feasible action is selected. The evaluated evolution policy uses failure detection and templates to instantiate the target and parameters, while learning the operator choice. Before either rule is applied, local preconditions and global controls define

$$\begin{aligned} \Omega(s_t) = \Omega_{\mathrm{lifecycle}}(s_t) \cap \Omega_{\mathrm{operator}}(s_t) \\ \cap\, \Omega_{\mathrm{ablation}}(s_t) \cap \Omega_{\mathrm{budget}}(s_t). \end{aligned} \tag{19}$$

The masked policy in Eq. 9 assigns zero probability outside this intersection. NoOp remains available as a total fallback, while a rejected topology-changing proposal leaves the active graph unchanged. Hard invariants are therefore enforced twice: cheap local masking avoids obviously illegal actions, and complete compilation validates the resulting candidate graph.

C.2 Rewrite Operator Contracts

Table 6 summarizes the graph transformation and interface contract of each action. Every topology-changing action is constructed on a graph copy and becomes visible only after the compiler accepts the complete candidate.

Recovery-node construction. Recovery nodes use deterministic template instantiation. For a failed target v_i, GPS applies a fixed recovery template to the original instruction and output contract,

$$\begin{aligned} I_i' &= \mathrm{RecoverTemplate}\big(I_i, \mathrm{Contract}(v_i)\big), \\ M_i' &= M_i, \qquad T_i' = \mathrm{Copy}(T_i), \\ C_i' &= \mathrm{Transfer}(C_i, F_i; i). \end{aligned} \tag{20}$$

Table 6: Semantic rewrite contracts. "Retire" keeps a replaced node in version history while excluding it from the active graph.

Operator	Graph transformation	Minimum precondition	State and interface rule
EXPAND	Replace the failed computation with one recovery node.	A typed target node and valid inherited bindings.	Inherits the target boundary, model, tools, contract, and transferable context.
SPLIT	Replace one node with parallel solver and checker units.	Target is neither the unique entry nor the unique output.	Both nodes inherit the target boundary and expose separate artifacts to each successor.
MERGE	Consolidate two active non-boundary nodes into one computation.	An explicit active peer distinct from the target.	Redirects both incident boundaries and compiles the merged interface before commit.
REWIRE	Route the complete failed-node boundary through a failure-aware replacement.	Target has at least one active successor.	Transfers typed failure metadata and redirects every incident active edge.
NOOP	Preserve the active graph.	Always legal.	Records the decision and continues without mutation.

The template instructs the node to redo the original operation from the task and available predecessor artifacts while preserving the exact output contract. The model binding is inherited from the target under the frozen routing configuration, and tool permissions are copied exactly. With persistent state enabled, Transfer copies the target's local context, adds a `recovery_for` identifier, and attaches typed failure metadata when required by the operator. In the corresponding ablation, it retains only immutable task fields such as the task and task type. A rewrite therefore changes program organization while preserving the model route, privilege set, and answer format.

EXPAND. Let $P_i = \mathrm{Pred}(v_i)$ and $S_i = \mathrm{Succ}(v_i)$. EXPAND creates one recovery node v_i' using Eq. 20, adds (p, v_i') for every $p \in P_i$ and (v_i', s) for every $s \in S_i$, removes the corresponding incident edges of v_i, and marks v_i as RETIRED. If v_i is a declared boundary, its entry or output identifier is moved to v_i'. The operation therefore introduces a fresh computation while preserving the observable interface of the failed node.

SPLIT. SPLIT creates v_i^{solve} and v_i^{check} from the same task contract and inherited bindings, assigns them recovery-solver and recovery-checker roles, and connects each node to every predecessor in P_i and every successor in S_i. The original incident edges are removed and v_i is retired. The two units are intentionally parallel: each successor receives both committed artifacts through dependency-scoped aggregation and can compare the independent recovery result with its check. Splitting the unique entry or output is masked because it would violate the single-boundary contract.

REWIRE. REWIRE constructs a failure-aware replacement v_i' whose transferred context includes F_i, then redirects the entire active boundary $P_i \times \{v_i\}$ and $\{v_i\} \times S_i$ through v_i'. The operator acts on the complete node boundary: if the failed node feeds several children, every child in S_i is reconnected to the replacement, while unrelated edges and the other predecessors of those children remain unchanged. All affected consumers therefore read from the same recovery path instead of mixing retired and replacement artifacts.

MERGE. For an active target v_u and explicit active peer v_v, neither of which is the unique entry or output, MERGE forms

$$P_{uv} = \big(\mathrm{Pred}(v_u) \cup \mathrm{Pred}(v_v)\big) \setminus \{v_u, v_v\}, \qquad S_{uv} = \big(\mathrm{Succ}(v_u) \cup \mathrm{Succ}(v_v)\big) \setminus \{v_u, v_v\}. \quad (21)$$

A deterministic template concatenates the two original instructions as separately named responsibilities. The merged node uses the runtime's active model route and the sorted union of both tool lists, which remains subject to the compiler's permission allowlist. Its context begins with the failed target's transferred context and adds only peer keys that are not already present, so target values resolve collisions. Input and output schemas are merged with target fields taking precedence, and security labels are unioned. Every edge incident to either target is redirected through the merged node, duplicate edges are collapsed, induced self-loops are removed, and both targets are retired.

Table 7: Compiler and runtime invariants enforced before a candidate graph becomes active.

Check	Acceptance condition
Endpoint validity	Every active edge references two existing nodes and contains no self-loop.
Acyclicity	$(\mathcal{V}^{\mathrm{act}}, \mathcal{E}^{\mathrm{act}})$ admits a topological ordering.
Unique boundaries	The active root set equals the declared entry node and the active sink set equals the declared output node.
Reachability	Every active node is reachable from the entry and can reach the output.
Node interface	Every active node has a nonempty instruction and a nonempty model route.
Lifecycle safety	No candidate graph is activated while an active node remains in the RUNNING state.
Runtime budget	The next transition remains within call, token, cost, retry, and rewrite caps.

The compiler then checks the resulting topology, interfaces, permissions, and reachability before commit.

Boundary preservation and atomicity. For a single-node replacement v_i', the external boundary satisfies

$$\mathrm{Pred}(v_i') = P_i, \qquad \mathrm{Succ}(v_i') = S_i, \qquad \sigma_i^+ = \mathrm{Retired}. \tag{22}$$

Split applies the same boundary to both recovery units. Compiler validation establishes global acyclicity, unique boundaries, and root-to-output reachability after each local transformation.

All operators follow the same state rule. Newly created nodes begin in PENDING with no committed output and zero attempts; unchanged node identifiers retain their compatible lifecycle state and artifacts; replaced nodes become RETIRED but remain in version history. Context needed by a new node is copied explicitly by Eq. 20 rather than recovered from hidden executor history. Candidate construction occurs on a graph copy, so none of these changes becomes visible until atomic cut-over.

C.3 COMPILER INVARIANTS AND ATOMIC CUT-OVER

The deterministic compiler enforces structural and interface invariants alongside the learned policy. Let $\mathcal{V}^{\mathrm{act}}$ and $\mathcal{E}^{\mathrm{act}}$ denote the active nodes and edges after excluding retired nodes. Acceptance requires the conjunction in Table 7.

Writing $\mathcal{C}_k$ for these predicates, the commit decision is

$$\begin{aligned} \chi_t &= \bigwedge_k \mathcal{C}_k(\widehat{\mathcal{G}}_{t+1}, X_t, B_t), \\ Y_{t+1} &= \begin{cases} \mathrm{Migrate}(\widehat{\mathcal{G}}_{t+1}, X_t), & \chi_t = 1, \\ Y_t, & \chi_t = 0. \end{cases} \end{aligned} \tag{23}$$

Candidate construction occurs on a graph copy. A rejected candidate therefore preserves the active node dictionary, edge set, boundary identifiers, and committed artifacts while returning the compiler error as policy feedback.

Validation complexity. Cycle detection, root/sink computation, and the two reachability traversals are linear in the active graph size. Hence a full compiler pass costs

$$\mathcal{O}\big(|\mathcal{V}^{\mathrm{act}}| + |\mathcal{E}^{\mathrm{act}}|\big). \tag{24}$$

Expand and Rewire edit the incident boundary of one target, Split duplicates that boundary, and Merge visits the incident edges of both targets. Although these edits scale only with the affected degrees, GPS still performs the global linear-time compiler pass because a locally plausible transformation may violate a global invariant.

C.4 RL POLICY TRAINING AND DETERMINISTIC DEPLOYMENT

Policy parameterization. The evaluated GPS policy learns which semantic operator to apply after runtime feedback exposes an evolution point. Failure detection supplies the target z_t, and the

operator-specific templates above instantiate ξ_t; the trainable policy maps the resulting state to a distribution over EXPAND, SPLIT, MERGE, REWIRE, and NOOP. Its compact state features are

$$\begin{aligned}\phi(s_t) &= [1, f_t, u_t, d_t],\\ f_t &= \mathbf{1}(\text{failure}_t), \qquad u_t = \text{clip}(\text{costPressure}_t, 0, 1),\\ d_t &= \min\big(\deg^+(z_t)/3, 1\big)\,.\end{aligned} \tag{25}$$

For each feasible operator $o \in \Omega(s_t)$, the learned logits and masked policy are

$$\widetilde{\pi}_\theta(o \mid s_t) = \frac{\exp(w_o^\top \phi(s_t))}{\sum_{o' \in \Omega(s_t)} \exp(w_{o'}^\top \phi(s_t))}, \qquad \theta = \{w_o\}_{o \in \mathcal{A}}. \tag{26}$$

In the evaluated configuration, failure detection and templates instantiate the target and parameters, while the learned weights w_o optimize operator selection from trajectory rewards. Masking and compilation independently enforce hard constraints.

RL objective and update. GPS optimizes the constrained return

$$J(\theta) = \mathbb{E}_{\tau \sim \widetilde{\pi}_\theta}\left[\sum_{t=0}^{T} \gamma^t r_t\right], \tag{27}$$

$$r_t = \alpha \Delta Q_t - \beta c_t - \eta \ell_t - \rho \mathbf{1}[\chi_t = 0] - \kappa \mathbf{1}[\text{failure}_t]. \tag{28}$$

When only terminal supervision is available, ΔQ_t is omitted and credit is derived from terminal task utility and resource use. The policy-gradient estimator is

$$\widehat{\nabla_\theta J} = \sum_{t=0}^{T} \nabla_\theta \log \widetilde{\pi}_\theta(a_t \mid s_t)\big(\widehat{G}_t - b(s_t)\big), \tag{29}$$

where $b(s_t)$ is the return baseline. Training repeatedly samples a feasible operator from $\widetilde{\pi}_\theta$, applies the corresponding rewrite to a graph copy, records the compiler outcome and task utility, and updates the same policy across trajectories. This retained optimization allows later decisions to reflect earlier success, cost, rejection, and failure signals.

Training and evaluation procedure. The policy lifecycle is

1. initialize θ and construct a training trajectory from an executable graph;
2. observe s_t, construct $\Omega(s_t)$, and sample $o_t \sim \widetilde{\pi}_\theta(\cdot \mid s_t)$;
3. instantiate the rewrite, compile the candidate, execute an accepted version, and record r_t;
4. update the retained parameters with Eq. 29 and continue across training trajectories;
5. select the trained checkpoint $\theta^\star$, freeze it, and use

$$o_t = \arg\max_{o \in \Omega(s_t)} \widetilde{\pi}_{\theta^\star}(o \mid s_t) \tag{30}$$

for formal evaluation.

The frozen masked argmax provides reproducible deployment of the learned checkpoint $\theta^\star$. RQ1–RQ4 evaluate this source-policy checkpoint, and RQ5 adapts it with the declared 16- or 32-example target-backbone set while keeping the graph representation, executor, templates, and compiler fixed.

D EXPERIMENTAL DETAILS

D.1 BENCHMARK CARDS

The nine benchmarks were selected to vary output type, reasoning depth, and the extent to which intermediate artifacts can affect later computation (Cobbe et al., 2021; Patel et al., 2021; Ling et al., 2017; Hendrycks et al., 2021b; Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021a; Yang et al., 2018; Chen et al., 2025). Table 8 states the evaluation role of each dataset. Evaluation uses benchmark-provided test instances and scorer semantics.

Table 8: Benchmark coverage and scoring. Common-5 is the subset used for the cost, ablation, and cross-backbone analyses.

Benchmark	Family	Evaluation role	Metric	Common-5
GSM8K	Grade-school mathematics	Multi-step numerical reasoning with a terminal numeric answer.	Accuracy	Yes
SVAMP	Arithmetic word problems	Tests robustness to controlled variations in simple arithmetic problem structure.	Accuracy	–
AQuA	Algebraic reasoning	Multiple-choice algebra problems paired with rationale-style solution structure.	Accuracy	–
MATH	Competition mathematics	Longer and more diverse competition-level derivations.	Accuracy	Yes
HumanEval	Code generation	Function synthesis from a signature and natural-language specification.	Pass@1	Yes
MBPP	Code generation	Short Python programs specified in natural language.	Pass@1	–
MMLU	Multi-domain reasoning	Multiple-choice knowledge and reasoning across heterogeneous subjects.	Accuracy	–
HotpotQA	Multi-hop QA	Requires combining evidence across multiple supporting facts.	Official score	Yes
BrowseComp+	Deep research	Fixed-corpus, multi-step retrieval and evidence integration.	Official task score	Yes

Code evaluation. Pass@1 counts a task as solved when the single generated candidate passes the benchmark tests. Evaluation executes extracted code in an isolated test harness and reports failures rather than repairing candidates with hidden tests. This keeps the metric comparable across single-agent, multi-agent, and graph-program organizations.

Cross-benchmark aggregation. All component scores are represented on the percentage scale before aggregation. For method m and benchmark set $\mathcal{D}$,

$$\begin{aligned} \mathrm{MacroAvg}(m;\mathcal{D}) &= \frac{1}{|\mathcal{D}|}\sum_{d\in\mathcal{D}} S_{m,d}, \\ \mathcal{D}_{\mathrm{Common5}} &= \{\mathrm{GSM8K}, \mathrm{MATH}, \mathrm{HumanEval}, \\ &\qquad \mathrm{HotpotQA}, \mathrm{BrowseComp{+}}\}. \end{aligned} \tag{31}$$

No benchmark-size weighting is applied. Thus, Common-5 measures the mean of five task metrics rather than the fraction of all pooled examples answered correctly.

D.2 Baseline Taxonomy

The baseline suite spans increasing degrees of organization, including graph pruning, graph optimization, task-conditioned topology design, selected-agent message graphs, and runtime rectification (Zhang et al., 2025b; Zhuge et al., 2024; Zhang et al., 2025c; Yun et al., 2026; Wang et al., 2026). Table 9 records the organization family and comparison role of each method.

D.3 Shared Controls and Comparison Blocks

The primary RQ1 block fixes the `DeepSeek-v4-pro` backbone, benchmark instances, decoding configuration, and evaluation scripts across systems. GPS configurations using GPT-5.5, Qwen3-14B, or a DeepSeek–Qwen combination form a complementary cross-backbone block that measures organization under heterogeneous model routes.

RQ2, RQ4, and RQ5 use Common-5 but answer different questions. RQ2 compares measured operating points under resource tiers; RQ4 modifies architectural capabilities while keeping the evaluation suite fixed; and RQ5 uses the manuscript-specified adaptation settings for a new backbone. Each analysis uses the scores produced by its corresponding protocol.

Table 9: Baseline taxonomy and its role in the controlled comparison.

Method	Organization family	Comparison role
Direct Prompting	Single model call	Backbone-only reference without explicit multi-step organization.
Zero-shot CoT	Single-agent reasoning	Adds an intermediate reasoning instruction without introducing a multi-agent topology.
Debate (3 agents)	Multi-agent deliberation	Tests whether repeated peer reasoning and aggregation suffice without executable program state.
Self-Consistency	Sample-and-aggregate	Aggregates independent reasoning paths, trading additional calls for output agreement.
AgentPrune	Pruned communication graph	Removes redundant spatial–temporal messages from an existing multi-agent pipeline.
GPTSwarm	Optimizable computational graph	Represents language-agent operations and their information-flow edges as optimization targets.
G-Designer	Task-conditioned topology	Generates a task-aware communication topology with a learned graph designer.
GoA	Selected-agent message graph	Selects a small agent set, constructs directed relevance paths, and aggregates graph messages.
MAS^2	Runtime-rectified MAS	Uses a generator–implementer–rectifier organization to configure and revise a target system from runtime demand.
GPS	Executable program graph	Maintains dependencies, committed artifacts, node state, semantic rewrites, and compiler-gated version transitions.

D.4 RQ1: Overall Effectiveness Protocol

RQ1 reports the official metric for each of the nine benchmarks. For a shared-backbone method m, the displayed arrow is a within-column difference from Direct Prompting,

$$\Delta_{m,d} = S_{m,d} - S_{\mathrm{Direct},d}. \tag{32}$$

Positive and negative arrows encode the arithmetic change under the same backbone. Best and second-best formatting is computed within the controlled shared-backbone block, while cross-backbone GPS rows report their absolute scores.

D.5 RQ2: Resource Accounting and Operating Points

For each evaluated run, the accounting layer records prompt tokens, completion tokens, successful and failed request attempts, and monetary charge under a frozen pricing snapshot. Total tokens and cost are

$$N_m^{\mathrm{total}} = \sum_{r \in \mathcal{R}_m} \left(N_r^{\mathrm{prompt}} + N_r^{\mathrm{completion}} \right), \tag{33}$$

$$C_m = \sum_{r \in \mathcal{R}_m} \left(p_{M(r)}^{\mathrm{in}} N_r^{\mathrm{prompt}} + p_{M(r)}^{\mathrm{out}} N_r^{\mathrm{completion}} \right), \tag{34}$$

where $\mathcal{R}_m$ contains all charged requests for method m and $M(r)$ is the model route used by request r. Request failures and retries remain in the cost ledger when the provider charges them. Because input and output rates can differ, token count and monetary cost need not induce the same ordering.

For observed point (C_i, S_i), configuration i is Pareto dominated when another observed configuration j satisfies

$$C_j \le C_i, \qquad S_j \ge S_i, \qquad (C_j, S_j) \ne (C_i, S_i). \tag{35}$$

The RQ2 figure connects the measured nondominated operating points and visualizes the observed performance–cost frontier.

Table 10: RQ4 ablation definitions. A disabled capability is removed from the runtime path rather than merely hidden from the result report.

Variant	Operational change
Full GPS	Persistent state, failure detection, semantic evolution, dynamic rewiring, and compiler validation are enabled.
w/o Dynamic Rewiring	REWIRE is removed from the selectable operator set; other enabled recovery operators remain available.
w/o Graph Evolution	Topology-changing recovery is disabled, leaving the initialized graph fixed during execution.
w/o Persistent Program State	Recovery nodes receive only immutable task-level context rather than mutable node-local failure context and reusable intermediate state.
w/o Persistent State and Rewiring	Combines the previous state removal with a disabled REWIRE operator.
Static Multi-Agent Graph	Uses a fixed graph organization without runtime program evolution.

D.6 RQ3: SYNTHETIC NODE-FAULT PROTOCOL

RQ3 evaluates GSM8K at per-node fault probabilities

$$p \in \{0, 0.1, 0.2, 0.3\}. \tag{36}$$

Each eligible node independently receives an injected-failure indicator before its output can commit. If an execution reaches k eligible nodes, the conditional probability of at least one injected event is

$$\Pr(\text{triggered} \mid k, p) = 1 - (1 - p)^k. \tag{37}$$

This identity explains why "30% per node" is not equivalent to "30% of tasks affected." Different methods may execute different numbers of eligible nodes, so RQ3 reports the declared node-level p rather than inferring a common task-level corruption rate.

Final accuracy uses all evaluated tasks at level p,

$$\text{Accuracy}(p) = \frac{N_{\text{correct}}(p)}{N_{\text{all}}(p)} \times 100\%, \tag{38}$$

whereas recovery is defined only for executions in which at least one injected event occurred,

$$\text{RecoveryRate}(p) = \frac{N_{\text{triggered and completed}}(p)}{N_{\text{triggered}}(p)} \times 100\%. \tag{39}$$

A recovered execution must produce a valid terminal answer within the same hard resource controls. Recovery rates are reported for evaluated intervention records, with unavailable entries displayed using an em dash.

D.7 RQ4: CONTROLLED COMPONENT ABLATIONS

RQ4 keeps the Common-5 suite and evaluation interface fixed and removes a declared capability. Table 10 gives the operational interpretation of every manuscript row.

For variant v, the reported decrease is

$$\Delta_v = S_{\text{Full GPS}} - S_v, \tag{40}$$

on the unweighted Common-5 macro-average. The combined state-and-rewiring ablation directly captures the interaction between persistent state and dynamic rewiring.

D.8 RQ5: CROSS-BACKBONE ADAPTATION

RQ5 uses the adaptation settings and values in Table 3. The source backbone is `DeepSeek-v4-pro`, the target is Qwen3-14B, and the evaluation metric is the Common-5 macro-average. The settings have the following manuscript-level meanings:

- **Zero-shot transfer** applies the source organization policy without target examples.
- **Policy-only adaptation** updates only the execution/evolution policy from 16 or 32 target-backbone examples while retaining the executor, compiler, prompts, and graph representation.
- **Full adaptation** updates the trainable GPS components using 32 target examples.
- **Target oracle** is the full-target reference reported in the manuscript table.

For adaptation setting a, the remaining oracle gap and recovered fraction of the initial zero-shot gap are

$$G_a = S_{\text{oracle}} - S_a, \tag{41}$$

$$\text{RecoveredGap}(a) = \frac{S_a - S_{\text{zero}}}{S_{\text{oracle}} - S_{\text{zero}}} \times 100\%. \tag{42}$$

Adaptation cost follows the manuscript table. RQ5 measures how efficiently the frozen source-policy checkpoint adapts to a new backbone under limited target supervision.