DESIGNING AND EVOLVING MULTI-AGENT SYSTEMS VIA A TYPED INTERMEDIATE REPRESENTATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Automatic design of LLM-based multi-agent systems (MAS) has largely taken one of two forms: a meta-agent writes each system as free-form code, which is expressive but hard to validate and entangles design quality with coding ability; or a search over prebuilt operators, which is reliable but usually settles for one design per task family. We take a middle route in which the meta-agent designs and an engine implements. For each query the meta-agent emits a typed intermediate representation (IR) that composes agents from interaction primitives and specifies each agent by role, objective, and instructions. A rule-based compiler deterministically lowers each valid IR into an executable MAS, making every design statically checkable, executor-independent, and fully traceable from architecture to outcome. An experience-guided evolutionary loop then mines successful strategies and failure lessons from development traces to refine the meta-agent's design policy, updating no model parameters and adding no test-time cost. With a lightweight open-source executor, our system attains the best average accuracy across four benchmarks spanning math and coding, exceeding the strongest of eight baselines by 9.3 points, and evolution improves every benchmark. Because execution is fully traced, gains and losses are attributable: evolution repairs coordination while residual errors localize to executor capability, making failures of automatically designed MAS inspectable. Code and data will be released.

1 INTRODUCTION

Large language model (LLM) multi-agent systems (MAS) tackle problems that outrun a single model by decomposing a task across specialized agents that plan, generate, critique, and coordinate (Wu et al., 2024; Hong et al., 2024; Li et al., 2023). But this leverage comes from design—the roles, prompts, and interaction topology—which is largely handcrafted and transfers poorly across tasks, motivating the automatic design of agentic systems (ADAS) (Hu et al., 2025), where a meta-agent or search procedure builds a MAS for each task. Once a machine takes over, the central question becomes how a design is **represented**: expressive enough to capture good systems, yet structured enough to be statically checkable, so designs run reliably and every outcome traces back to a specific decision.

Existing methods differ mainly in this representation and fall into two families, each gaining one strength at the other's expense. The first has a meta-agent emit each system as free-form code (Hu et al., 2025; Ye et al., 2025; Gao et al., 2025): maximally expressive, but generated programs often fail to run, any failure may reflect weak design, weak coding, or an interface mismatch, and reliable generation usually demands costly training of the generator (Ye et al., 2025; Gao et al., 2025). The second searches over a fixed library of prebuilt operators (Zhuge et al., 2024; Zhang et al., 2025c; Zhou et al., 2026; Zhang et al., 2025a): reliable, but bounded by components fixed in advance and typically committed to one design per task family, unable to author a new agent or adapt to queries that differ widely within a family. Expressivity and reliability are thus played off against each other, and neither route makes failures easy to attribute, even though manual audits trace most MAS failures to specification and coordination defects (Cemri et al., 2025), precisely the errors an automatic designer can silently introduce.

As Figure 1 (c) shows, we take a middle route inspired by how a compiler separates a front-end from a back-end: the meta-agent **designs** and a deterministic engine **implements**. Generating com-

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

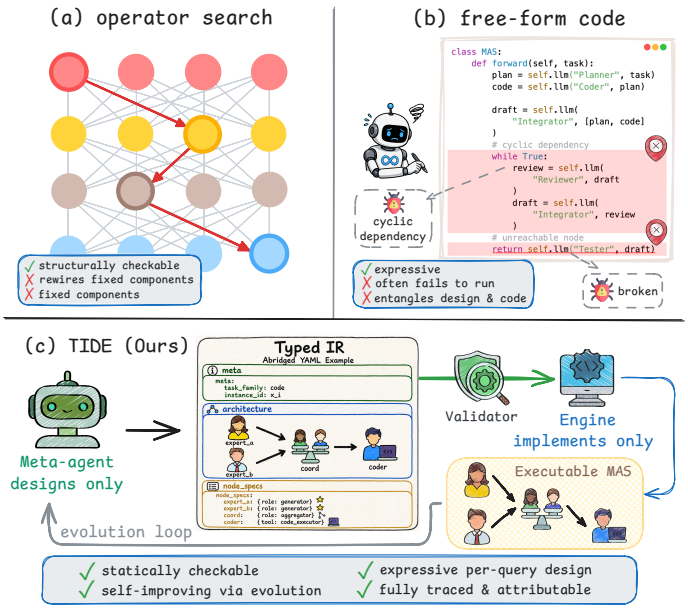


Figure 1: Positioning of TIDE. (a) Operator search is statically checkable but rigid; (b) free-form code is expressive but fragile. (c) TIDE separates MAS design from implementation like a compiler's front-end and back-end, staying expressive yet statically checkable through a typed IR that a deterministic engine validates, compiles, and traces.

plete framework code forces the meta-agent to handle low-level syntax and framework APIs, which is harder and more error-prone than focusing on the collaboration architecture and agent design, where the LLM is strongest at organizing work: deciding who does what, in what order, and how partial results combine. We propose TIDE, a **T**yped **I**ntermediate representation for **D**esigning and **E**volving multi-agent systems, which realizes this idea: the meta-agent expresses a design as an explicit, checkable artifact while the engine handles implementation. This decouples the design and coding abilities that free-form code entangles, and escapes the rigidity of search- and graph-based methods, which fix the components in advance and only optimize the wiring among them, whether by searching over prebuilt operators, pruning a fully connected agent graph, or learning the best connections on a predefined graph. Because the IR predefines only an extensible registry of interaction primitives, roles, and tools while the meta-agent authors the agents that draw on them, designs stay expressive yet statically checkable, and full tracing makes every outcome attributable to the design choice behind it.

On this interface the meta-agent gives each agent a role and tools from these registries, writes its objective and instructions in natural language, and composes the agents into an acyclic architecture from interaction primitives that carry execution semantics but no implementations or prompts, like a language's control constructs. A static validator rejects malformed designs; a deterministic compiler with no LLM lowers each valid design into executable control flow and communication contracts, binding each agent to an executor model only at instantiation; and the runtime traces every step. This traceability makes design-level learning possible: an MAS outcome can be attributed to specific, reusable design decisions. On top of the engine we therefore add an **experience-guided evolution loop**. A textual critic contrasts successful and failed runs on a development set, turns recurrent failure modes and their counter-patterns into reusable design experience, and folds it into the meta-agent's design policy, keeping a change only when development accuracy holds or improves. Such training-free self-improvement from execution feedback has proven effective for an individual agent's memory or context (Ouyang et al., 2026; Zhang et al., 2026). The typed IR carries it to the system's architecture, where a deliberately simple loop turns traces into better MAS designs and amortizes the gains into the design policy, so each test query still receives a single generated design and a single compiled execution.

Evaluated with a lightweight open-source executor on four benchmarks spanning math and coding, the evolved system reaches an average accuracy of 78.0, exceeding the strongest of eight base-

lines, including ADAS, MAS-GPT, and MASS, by 9.3 points. Evolution improves every benchmark, adding 7.2 points on average over the same engine before evolution. Because execution is fully traced, these gains are attributable: evolution prunes redundant agents, consolidates evidence-preserving topologies, and repairs answer arbitration, so that on AIME, cases where a node states the gold answer but the system then discards it fall from 10 to 1, while residual errors localize to executor capability. This attributability makes automated MAS design inspectable.

Our contributions are threefold: (1) a typed IR for expressive, statically checkable per-query MAS design that separates architecture from implementation; (2) a deterministic compiler producing traceable, executor-independent systems, coupled with experience-guided evolution that improves design policies without weight updates or additional test-time cost; and (3) the best average accuracy across four benchmarks against eight baselines, with trace-based analyses showing that evolution repairs coordination while residual errors localize to executor capability.

2 Related Work

LLM multi-agent systems. LLM agents solve tasks together through conversation, role specialization, debate, and shared workspaces (Wu et al., 2024; Hong et al., 2024; Li et al., 2023; Du et al., 2023). Designing them by hand is labor-intensive, demands careful tuning of prompts and interaction patterns, and transfers poorly to novel tasks. Their behavior hinges on coordination: trace audits attribute most failures to specification and coordination defects (Cemri et al., 2025), making the system's design, not only the model, a primary target for improvement.

Automatic agentic system design. This motivates the automated design of agentic systems (ADAS) (Hu et al., 2025), where a meta-agent or a search procedure builds a system per task and methods differ mainly in what they generate. Code-space methods pursue maximal expressivity by emitting executable code or code workflows, and some train per-query generators to produce them (Zhang et al., 2025c; Ye et al., 2025; Gao et al., 2025). The resulting programs are fragile, entangle design quality with low-level coding so failures are hard to attribute, and are expensive to obtain when a generator must be trained. Structure-space methods trade expressivity for reliability, fixing a vocabulary of agents or operators and searching or learning only the communication topology among them (Zhuge et al., 2024; Zhou et al., 2026; Shang et al., 2025; Zhang et al., 2025a;b; Li et al., 2026; Liu et al., 2024). This is robust but bounded by the predefined pool, often relies on trainable graph components, and usually commits to a single design per task family instead of adapting to individual queries. Our IR combines their strengths: the meta-agent authors the components in a typed language, keeping designs expressive yet statically checkable; a deterministic compiler lowers each into a traceable executable; and learning is amortized offline, so every query needs only one generation and one compiled run.

Feedback-driven optimization. Prior work improves an LLM system from its own feedback without weight updates, differing in where the feedback acts. At the output level, a model critiques and revises its own answers within a single episode (Wei et al., 2022; Madaan et al., 2023). At the prompt level, scores or textual gradients tune the instructions of an otherwise fixed pipeline (Yang et al., 2024; Khattab et al., 2024; Yuksekgonul et al., 2024). At the memory level, lessons distilled from past episodes are stored and retrieved to steer later runs, an increasingly popular form of training-free test-time adaptation (Shinn et al., 2023; Zhao et al., 2024; Ouyang et al., 2026; Zhang et al., 2026). Our experience-guided evolution acts at the design level: it turns execution traces into reusable experience that reshapes the architectures and agent specifications the meta-agent proposes for later queries.

3 Method

3.1 Problem Formulation

Let x be a task instance from task family b, G_ϕ a meta-agent, and E_ψ an executor assignment binding an LLM to each agent (possibly a different model per agent). Given a fixed system prompt π (the IR language and general design rules) and a task-family design prompt p_b in the user prompt, the

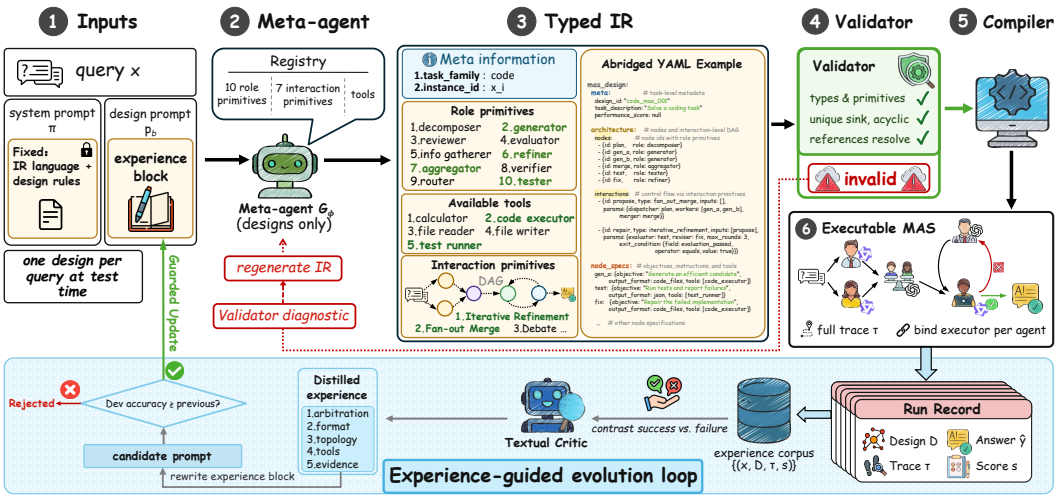


Figure 2: Overview of TIDE. *Top:* from a registry of roles, interaction primitives, and tools the meta-agent emits a typed IR; a static validator returns malformed designs for regeneration, and a deterministic compiler with no LLM lowers valid designs into a traced executable MAS. *Bottom:* a critic distills traces into reusable experience that refines the meta-agent's designs, kept only when accuracy holds or improves, at no weight-update or test-time cost.

meta-agent emits a system description D that the engine compiles and runs into an answer $\hat{y}$ and a complete trace τ:

$$\begin{aligned} D &= G_\phi(x, p_b;\, \pi), \\ (\hat{y}, \tau) &= \mathrm{Run}(\mathrm{Compile}(D, E_\psi), x). \end{aligned} \tag{1}$$

Requiring D to lie in a typed design language $\mathcal{L}_{\mathrm{IR}}$ turns generation into constrained program synthesis: the meta-agent sets the collaboration structure and agent specifications, the engine owns implementation, scheduling, and state propagation, and *typed* means every field has a declared type, every reference resolves, and every constraint is checked before execution. Experience-guided evolution later updates p_b alone, leaving π and the engine fixed. Figure 2 shows the full pipeline and the evolution loop.

3.2 A TYPED INTERMEDIATE REPRESENTATION FOR MAS

Structure. Concretely, an IR is a single YAML configuration file with three top-level components: `meta` holds task metadata, `architecture` declares the agents and their interaction blocks, and `node_specs` maps each agent to its persona, objective, instructions, output contract, tools, and retry bound. Its design vocabulary has three tiers.

Vocabulary. We adopt a compact set of general, widely-used interaction primitives and roles (Wu et al., 2024; Hong et al., 2024; Du et al., 2023; Madaan et al., 2023; Zhou et al., 2026) for two reasons: they are expressive enough for the designs we target, and they keep experimentation and validation simple. The registry can still be extended or modified as needed. **Role primitives** state what an agent is for. The registry lists ten (`decomposer`, `generator`, `reviewer`, `evaluator`, `refiner`, `info gatherer`, `aggregator`, `verifier`, `router`, `tester`), each with functional semantics and built-in safeguards: judgment roles write only to control channels, recording verdicts and feedback without touching the answer, so a critic can never silently discard or degrade a correct result. **Interaction primitives** carry the control-flow semantics (Table 1). Each block declares an identifier, a primitive type, typed parameters, and its upstream blocks. Omitting inputs chains a block to its predecessor, an empty list makes a source, and multiple inputs form a fan-in. The graph must be acyclic, with a single sink whose output is the final answer. **Interaction slots** are the positions inside an interaction (`dispatcher`, `worker`, `merger`, `evaluator`, `reviser`). The slot, not the agent, fixes what shared state is read and written at run time.

Table 1: The seven interaction primitives. Each carries the control flow and the communication structure among agent nodes; multiple blocks compose into an interaction-level DAG.

Primitive	Execution semantics
Handoff	One agent transfers its result to another.
Sequential chain	Agents execute in a declared order.
Iterative refinement	An evaluator and reviser alternate until a typed exit condition or round bound.
Fan-out merge	A dispatcher launches parallel workers and a merger aggregates indexed outputs.
Debate	Parallel participants propose arguments and a moderator returns a complete answer.
Conditional route	A router selects one labeled branch with a typed fallback.
Vote	Parallel voters are reduced by majority or merge aggregation.

Constraint and expressiveness. The IR deliberately separates what is fixed from what stays open. What governs coordination is fixed: the interaction primitives, along with the sets of roles and tools agents draw from, are given in advance. What each agent is and does is open: for each query the meta-agent selects its role and tools from those sets and authors its persona, objective, instructions, and output contract in natural language. Constraint therefore applies to control flow and communication, where coordination errors arise, and expressiveness to what each agent does, where design judgment matters. The split also makes designs correct by construction: because node specifications name no channels, the compiler derives every read and write set from an agent's slot and the primitive's communication contract, ruling out a whole class of coordination faults (reading stale state, writing to the wrong place, or dropping a teammate's output) before a design ever runs.

Extensibility and validation. The vocabulary is a tunable surface the designer can grow, and extending it is cheap: a new primitive takes one registry entry and one lowering rule, and the meta-agent's design instructions are rendered from that registry, so prompt, validator, and compiler stay in sync. Before execution, a schema validator checks types, primitive membership, identifier uniqueness, complete node specifications, reference resolution, bounded rounds, DAG acyclicity, node reachability, and a unique sink, returning any violations to the meta-agent as one readable diagnostic that it must repair with a complete replacement IR. This repair loop runs on well-formedness alone, independent of any task score or gold answer.

3.3 Deterministic Compilation and Traced Execution

Compiler passes. For a valid design, the compiler runs three rule-based passes. **(1) Schema.** It derives a blackboard schema from the IR: named channels with value types and merge rules, plus each agent's per-interaction read and write sets. A pure function of the IR, this schema is both the communication contract the lowered blocks implement and an auditable specification for checking execution and traces. **(2) Lowering.** It maps each primitive to a control block with schema-fixed read and write sets. Parallel writers fill slots by declared order, so a merged result is deterministic regardless of which agent finishes first. It also enforces the role safeguards: judgment outputs go to typed control channels, and only the final-answer step writes the deliverable, so a critic cannot overwrite the answer. **(3) Scheduling.** It builds a stable topological schedule in which dependency-ready blocks run in parallel only when they share neither an agent nor a file workspace, and the unique sink maps to the final-answer channel.

Instantiation and execution. Compilation is a pure structural transformation and invokes no LLM. Executor models are bound per agent only when node functions are built, so a validated design can be re-instantiated on a different executor, or mix models across agents; our main experiments give every agent the same model for simplicity and fair comparison. At run time, each node prompt is assembled from the agent's persona, objective, instructions, output contract, upstream context, and tools. Scheduler-facing fields such as routing, evaluation, and moderation are parsed into typed control contracts that the scheduler acts on, and any registered tools are exposed through fenced calls whose results return to the node for a bounded number of rounds.

Full tracing. Every state transition is traced: scheduler waves, interaction inputs and outputs, node prompts and responses, parsed control values, tool results, token consumption, and final state. Shared node, interaction, and channel identifiers link the declared design D, its compiled execution, and its outcome, so any result traces back to the design choice that produced it.

3.4 Experience-Guided Design Evolution

Experience extraction. The task-family prompt starts minimal: $p_b^{(0)}$ carries only the output contract for family b, since the IR language and general rules live in the fixed system prompt π. At each evolution round k, running the current prompt $p_b^{(k)}$ on a development set $\mathcal{D}_b^{\mathrm{dev}}$ yields an experience corpus

$$\mathcal{E}_b^{(k)} = \{(x_i, D_i^{(k)}, \tau_i^{(k)}, s_i^{(k)})\}_{i \in \mathcal{D}_b^{\mathrm{dev}}}, \tag{2}$$

with s_i from the benchmark evaluator. A textual critic contrasts successful and failed traces to distill a compact diagnosis $g_b^{(k)}$; it sees the designs, traces, predictions, and scores, but the extracted experience may not contain gold answers, instance-specific equations, hidden tests, or candidate solutions.

Guarded update. An evolution step rewrites the experience block into a candidate $\tilde{p}_b^{(k)} = \mathrm{Evolve}(p_b^{(k)}, g_b^{(k)})$; writing $S(p)$ for the mean development accuracy under prompt p, we accept it only if it does not regress:

$$p_b^{(k+1)} = \begin{cases} \tilde{p}_b^{(k)} & \text{if } S(\tilde{p}_b^{(k)}) \geq S(p_b^{(k)}), \\ p_b^{(k)} & \text{otherwise.} \end{cases} \tag{3}$$

The rewrite must capture reusable, effective design strategies rather than memorized corrections. In practice, the extracted experience falls into five recurring categories: (i) when additional agents add independent evidence, (ii) which topologies preserve or destroy the main answer, (iii) how to arbitrate disagreement, (iv) when tools provide objective evidence, and (v) how to preserve task interfaces and terminal output formats. Each accepted prompt is kept as an immutable version.

Selection and deployment. This resembles evolution more than gradient descent: traces define the environment, the critic proposes a textual mutation, and measured outcomes select the next prompt, computing no gradients, updating no parameters, and never touching the compiler. At test time the selected experience block is appended to the design prompt before IR generation, so every system, initial or evolved, uses one design attempt per query and differs only in the learned design policy. Algorithm 1 summarizes the full procedure.

Algorithm 1 Experience-guided design evolution and one-shot deployment

Require: $\mathcal{D}_b^{\mathrm{dev}}$, test task x, initial prompt $p_b^{(0)}$, G_ϕ, E_ψ, critic C, evaluator A_b, rounds K

for $k = 0, \ldots, K$ **do**
  $\mathcal{E}_b^{(k)} \leftarrow \mathrm{EvaluatePrompt}(p_b^{(k)}, \mathcal{D}_b^{\mathrm{dev}}; G_\phi, E_\psi, A_b)$
  if $k < K$ **then**
    $g_b^{(k)} \leftarrow \mathrm{ExtractExperience}(C, \mathcal{E}_b^{(k)})$
    $p_b^{(k+1)} \leftarrow \mathrm{EvolvePrompt}(p_b^{(k)}, g_b^{(k)})$
  end if
end for
$p_b^\star \leftarrow \mathrm{Select}(\{p_b^{(k)}, \mathcal{E}_b^{(k)}\}_{k=0}^{K})$
$D \leftarrow \mathrm{GenerateRepairIR}(G_\phi, x, p_b^\star)$
$(\hat{y}, \tau) \leftarrow \mathrm{Run}(\mathrm{Compile}(D, E_\psi), x)$
return Answer $\hat{y}$, typed design D, and trace τ

4 Experiments

4.1 Experimental Setup

Benchmarks. We evaluate on four benchmarks spanning two task families. Math reasoning uses AIME (Balunović et al., 2026) (pooling the 2025 and 2026 contests) and MATH (Hendrycks et al., 2021), and coding uses HumanEval+ (Chen et al., 2021; Liu et al., 2023) and MBPP (Austin et al., 2021). Math answers are graded by symbolic equivalence, and coding solutions by executing the generated Python against each benchmark's test suite. We report accuracy, and pass@1 for coding, in percentage points.

Baselines. We compare against eight baselines in two groups. Three run a single agent: direct prompting (Vanilla), chain-of-thought prompting (CoT; Wei et al., 2022), and iterative self-correction (Self-Refine; Madaan et al., 2023). Five are automatic designers that construct a MAS for the task: DyLAN (Liu et al., 2024), GPTSwarm (Zhuge et al., 2024), ADAS (Hu et al., 2025), MAS-GPT (Ye et al., 2025), and MASS (Zhou et al., 2026).

Implementation details. Under our compute budget, the meta-agent is Qwen3-32B (Yang et al., 2025) and every node runs the lightweight Qwen3-4B-Instruct-2507 executor, shared across nodes for efficiency and fair comparison; both models use a 16,384-token limit at temperature 0.6, dropping to 0.3 for code. The meta-agent equips each executor agent with tools from a registry of a calculator, a code executor, file reader and writer, and a test runner. During evolution, a Qwen3-32B critic distills experience from scored traces on a held-out development split of each task family to update the design prompt, with split proportions and further details in the Appendix.

4.2 Main Results

Table 2: Main accuracy (%) across four benchmarks. AIME pools the AIME 2025 and 2026 contests. TIDE w/o and w/ Evolution differ only in the task-family design prompt. Best values are boldfaced; second-best values are underlined.

Method	Executor Model	Math Reasoning		Coding		Average
		AIME	MATH	HumanEval+	MBPP	
Vanilla	Qwen3-4B	33.3	82.1	81.7	53.9	62.8
CoT (Wei et al., 2022)	Qwen3-4B	<u>48.4</u>	82.9	86.0	53.5	67.7
Self-Refine (Madaan et al., 2023)	Qwen3-4B	40.0	<u>91.9</u>	85.4	56.7	68.5
DyLAN (Liu et al., 2024)	Qwen3-4B	38.4	91.7	85.4	57.1	68.2
GPTSwarm (Zhuge et al., 2024)	Qwen3-4B	36.7	79.3	84.8	54.5	63.8
ADAS (Hu et al., 2025)	Qwen3-4B	40.0	82.8	83.5	45.1	62.9
MAS-GPT (Ye et al., 2025)	Qwen3-4B	36.7	89.2	<u>88.4</u>	53.1	66.9
MASS (Zhou et al., 2026)	Qwen3-4B	<u>48.4</u>	73.0	81.7	71.5	68.7
TIDE w/o Evolution	Qwen3-4B	35.0	90.0	84.8	<u>73.4</u>	<u>70.8</u>
TIDE w/ Evolution	Qwen3-4B	**51.7** (+16.7)	**94.0** (+4.0)	**90.9** (+6.1)	**75.2** (+1.8)	**78.0** (+7.2)

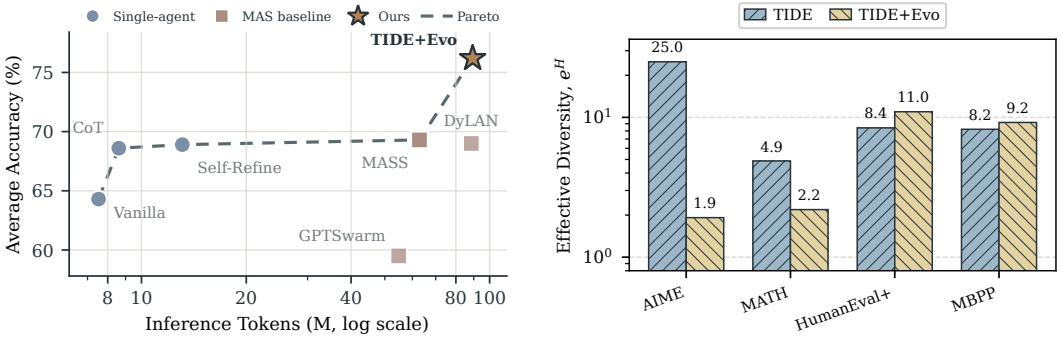


Figure 3: Analysis of the deployed TIDE systems. (a) Accuracy–cost trade-off on the main benchmarks: TIDE+Evo sits at the top of the Pareto frontier with the best accuracy. (b) Effective design diversity $\exp(H)$ per task before (TIDE) and after (TIDE+Evo) evolution; evolution concentrates the design space on AIME and MATH and broadens it on the coding tasks.

Table 2 reports accuracy across the four benchmarks. TIDE+Evo attains the best average accuracy at 78.0, exceeding the strongest baseline (MASS at 68.7) by 9.3 points and ranking first on all four task columns, leading every math and coding benchmark. Two points stand out. First, the advantage does not solely rest on evolution: without it the engine already averages 70.8 and tops every baseline, so a capable meta-agent that expresses designs in a typed IR and compiles them deterministically puts its structural priors to work directly, and much of the gain comes from the design paradigm rather than search or training. Second, experience-guided evolution adds 7.2 points and improves every benchmark, showing that in a checkable and traceable representation even a

deliberately simple experience loop yields consistent gains. The baselines also trade strength across families, with MASS leading on MBPP (71.5) but weakest on MATH (73.0) and Self-Refine and DyLAN topping MATH yet dropping to about 57 on MBPP, whereas TIDE stays strong on both.

4.3 SYSTEM RELIABILITY AND EFFICIENCY

The typed IR and deterministic compiler make the executable design space highly reliable. Across the 3,448 executions behind Table 2, only one produces an invalid IR (0.03%), and every other design compiles and runs, because the validator rejects or repairs malformed designs before execution. This reliability also matters downstream, letting experience-guided evolution and our failure analysis operate on genuine executions. On cost, TIDE+Evo is Pareto-optimal in accuracy versus token budget (Figure 3(a)), reaching the highest accuracy among methods with measured consumption; DyLAN spends a comparable budget yet scores over 9 points lower, and cheaper baselines such as CoT, Self-Refine, and MASS occupy lower-accuracy points on the same frontier.

4.4 HOW DOES THE DESIGN POLICY EVOLVE

Because the IR normalizes every generated system, we can compare architectures without program analysis. We map each design to a coarse skeleton consisting of its ordered primitive types, dependency pattern, and node counts. Let p_s denote the empirical frequency of skeleton s, and measure diversity by

$$H = -\sum_s p_s \log p_s, \qquad D_{\text{eff}} = \exp(H).$$

Here D_{eff} is the effective number of equally frequent skeletons: if K skeletons occur uniformly then $D_{\text{eff}} = K$, so lower values indicate that the design policy concentrates on fewer architectures. We compute these quantities on the same executor runs behind Table 2, and, as Figure 3(b) shows, evolution moves the effective diversity D_{eff} of each family in a task-specific direction. On the math families it concentrates the policy on designs that produce several independent candidates and aggregate them, with multi-upstream fan-in rising from about 30% to over 80% on AIME and from 9.6% to 51.0% on MATH; because these problems have verifiable answers, generating several candidates and aggregating them works well across queries, so the policy converges onto this single strong structure and its effective diversity drops, consistent with the large gains on math. On the coding families it instead increases diversity: coding queries vary more from one to the next, and interface, control-flow, and edge-case failures each need a different fix, so no single design fits all and the policy keeps a wider mix of architectures. Detailed statistics on the interaction primitives behind these skeletons are in the Appendix. In short, a guarded textual critic, updating no weights, moves each family's design distribution toward a structural prior suited to the task, and these shifts track the per-family accuracy gains in Table 2.

4.5 FAILURE ANATOMY

To understand *why* automatic designs fail, we use a strong LLM to trace a sample of failed runs back to the node where the run first leaves a correct path. This exposes two kinds of failure. In a *capability failure* the correct answer never appears at any node, so the executor is the limit. In a *design failure* some node reaches the correct answer but the system then loses it, and these recur in four modes. **(1) Specification drift.** The design misreads or rewrites the task before solving even begins, for instance inventing a constraint the problem never stated or changing what a function should return, so every later node works from the wrong problem. **(2) Error propagation.** An early node commits to a flawed plan or wrong intermediate result, and later nodes build on it without ever checking it, so one early mistake compounds through the system. **(3) Verification theater.** A reviewer or tester declares success with no executable evidence, so it never actually verifies anything. On coding tasks, for instance, a node is given a code executor but never calls it, and the design only looks verified while nothing is run. **(4) Post-solution degradation.** A correct candidate is overwritten during aggregation or buried under JSON or commentary at the sink, so a right answer is found and then thrown away.

Evolution mainly repairs these design failures. On AIME, post-solution degradation, where a node states the gold answer but the terminal step returns a different one, drops from 10 cases to 1, leaving executor capability as the main residual. Trace feedback therefore teaches the meta-agent restraint,

Table 3: Executor scaling accuracy with a fixed meta-agent. Best values are boldfaced; second-best values are underlined.

Benchmark	Qwen3-8B	Qwen3-14B	Qwen3-32B
AIME 2025	16.7	<u>20.0</u>	**26.7**
AIME 2026	10.0	<u>20.0</u>	**26.7**
GPQA	42.0	<u>43.8</u>	**48.4**
MBPP	55.2	<u>71.8</u>	**73.0**
LiveCodeBench	26.3	<u>29.1</u>	**30.3**
Average	41.2	<u>48.2</u>	**50.6**

Table 4: Meta-agent scaling accuracy with the Qwen3-4B-Instruct-2507 executor fixed. Best values are boldfaced; second-best values are underlined.

Benchmark	Qwen3-8B	Qwen3-32B	Qwen3-235B
AIME 2025	40.0	**46.7**	<u>43.3</u>
AIME 2026	33.3	**56.7**	**56.7**
GPQA	46.9	<u>48.2</u>	**54.0**
MBPP	72.8	<u>73.4</u>	**76.4**
LiveCodeBench	30.3	**34.9**	<u>32.0</u>
Average	44.7	<u>52.0</u>	**52.5**

pointing future engines toward evidence-gated collaboration, execution-grounded verification, and artifact-preserving transitions. Further examples and details are in the Appendix.

4.6 Executor Scale and Portability

The IR/compiler boundary lets one design language bind to different executors. Holding the Qwen3-32B meta-agent, design prompt, compiler, scorer, and one-shot budget fixed, we bind generated systems to non-thinking Qwen3-8B, 14B, or 32B executors, with executor identity hidden from the meta-agent. Here we evaluate on AIME, GPQA (Rein et al., 2024), MBPP, and add a recent LiveCodeBench (Jain et al., 2025) subset (v6 incremental split, 175 problems from January–April 2025 contests) as a contamination-resistant coding test. Larger executors help on average (41.2, 48.2, and 50.6 for 8B, 14B, and 32B) and improve every task family as they scale (Table 3), though at different rates. The gains are largest where the executor's own capability is the bottleneck, as on AIME, which rises with executor size in both years, and on GPQA. They taper where the design already captures most of the value, as on MBPP, which jumps 16.6 points from 8B to 14B and then adds 1.2 at 32B. Because the meta-agent and its design prompt stay fixed while the executor is hidden from it, the same design policy transfers across executors and turns a stronger executor directly into higher accuracy.

4.7 Meta-Agent Scale and the Design Prior

With the Qwen3-4B-Instruct-2507 executor, compiler, prompts, scorer, and one-shot TIDE procedure fixed, we vary only the meta-agent across Qwen3-8B, Qwen3-32B, and Qwen3-235B. Each meta-agent samples its own IR, so the comparison spans three design policies, one per model scale, each free to synthesize its own systems. Scaling the meta-agent yields diminishing marginal returns: the average climbs sharply from 44.7 at 8B to 52.0 at 32B, but only inches to 52.5 for the far larger 235B (Table 4), so a mid-sized meta-agent already suffices. The gains are also selective. Beyond 32B, the 235B designer improves the knowledge- and code-oriented columns where design judgment pays off, lifting GPQA to 54.0 and MBPP to 76.4, while it stays flat or slightly lower on AIME and LiveCodeBench, where the 4B executor's own capability is the binding constraint. Meta-agent scale therefore acts mainly on the design prior, the kind of system the policy tends to produce, and it improves accuracy only on tasks where better orchestration helps. We detail these structural shifts in the Appendix.

5 Conclusion

We presented a method for automatic MAS design that separates architectural decisions from implementation code. A meta-agent writes a typed IR, a deterministic compiler lowers it to a traced runtime, and experience extraction evolves the design policy from execution traces without updating model parameters. The resulting system improves average performance across four benchmarks while making both gains and failures inspectable. More broadly, the results suggest that progress in automatic MAS design depends not only on searching larger spaces, but on representing designs so that validity, execution, coordination, and executor capability can be measured independently.

AI USE STATEMENT

We used generative AI tools for literature searches, language editing, and assistance with selected parts of the method implementation. AI-assisted outputs were reviewed and revised by the authors. The authors take responsibility for the final implementation, experimental results, and manuscript.

REFERENCES

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

Mislav Balunović, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. Matharena: Evaluating llms on uncontaminated math competitions, 2026. URL https://arxiv.org/abs/2505.23281.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent llm systems fail?, 2025. URL https://arxiv.org/abs/2503.13657.

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, 2021. URL https://arxiv.org/abs/2107.03374.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate, 2023. URL https://arxiv.org/abs/2305.14325.

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, 2025. URL https://arxiv.org/abs/2504.15257.

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 *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021. URL https://openreview.net/forum?id=7Bywt2mQsCe.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, volume 2024, pp. 23247–23275, 2024.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=t9U3LW7JVX.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=chfJJYC3iL.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=sY5N0zY5Od.

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 *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=3IyL2XWDkG.

Shiyuan Li, Yixin Liu, Qingsong Wen, Chengqi Zhang, and Shirui Pan. Assemble your crew: Automatic multi-agent communication topology design via autoregressive graph generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 23142–23150, 2026.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in neural information processing systems*, 36:21558–21572, 2023.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. A dynamic llm-powered agent network for task-oriented agent collaboration. In *First Conference on Language Modeling*, 2024.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=jL7fwchScm.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=Ti67584b98.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. Agentsquare: Automatic llm agent search in modular design space. In *International Conference on Learning Representations*, volume 2025, pp. 3841–3865, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

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 conversations. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BAakY1hNKS.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 12028–12068, 2024.

Rui Ye, Shuo Tang, Rui Ge, Yaxin Du, Zhenfei Yin, Siheng Chen, and Jing Shao. MAS-GPT: Training LLMs to build LLM-based multi-agent systems. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=3CiSpY3QdZ.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic "differentiation" via text, 2024. URL https://arxiv.org/abs/2406.07496.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, LEI BAI, and Xiang Wang. Multi-agent architecture search via agentic supernet. In *Forty-second International Conference on Machine Learning*, 2025a. URL https://openreview.net/forum?id=imcyVlzpXh.

Guibin Zhang, Yanwei Yue, Xiangguo Sun, Guancheng Wan, Miao Yu, Junfeng Fang, Kun Wang, Tianlong Chen, and Dawei Cheng. G-designer: Architecting multi-agent communication topologies via graph neural networks. In *Forty-second International Conference on Machine Learning*, 2025b. URL https://openreview.net/forum?id=LpE54NUnmO.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025c.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=eC4ygDs02R.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan O Arik. Multi-agent design: Optimizing agents with better prompts and topologies. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=I05H9RUzHB.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=uTC9AFXIhg.

SUPPLEMENTARY MATERIAL

A IMPLEMENTATION DETAILS

Design-to-execution workflow. For each query, the meta-agent receives the task, its task-family design prompt, and the fixed typed IR specification, and emits one YAML design. A schema validator checks role and interaction names, required parameters, node references, graph acyclicity, reachability, and a unique terminal interaction. A deterministic compiler then instantiates the executor nodes, lowers each interaction primitive, resolves interaction dependencies, and runs the resulting graph, invoking no LLM inside the compiler. Finally, the benchmark adapter materializes the terminal artifact and invokes the benchmark-specific scorer.

Models and inference settings. The meta-agent is Qwen3-32B and the executor nodes run Qwen3-4B-Instruct-2507. Both use a 16,384-token completion limit at temperature 0.6, dropping to 0.3 on coding tasks. Every reported benchmark execution uses one sampled design and one execution. An invalid YAML or IR is repaired by feeding the validator diagnostic back to the meta-agent for bounded structural repair, which never exposes a gold answer or evaluator score. Table 5 lists the exact public checkpoints used in the main, executor-scaling, and meta-agent-scaling experiments.

Table 5: Model checkpoints used in the main, executor-scaling, and meta-agent-scaling experiments.

Provider/ Series	Model Name	Model Link
Qwen	Qwen3-4B-Instruct-2507	`https://huggingface.co/Qwen/Qwen3-4B-Instruct-2507`
	Qwen3-8B	`https://huggingface.co/Qwen/Qwen3-8B`
	Qwen3-14B	`https://huggingface.co/Qwen/Qwen3-14B`
	Qwen3-32B	`https://huggingface.co/Qwen/Qwen3-32B`
	Qwen3-235B-A22B-Thinking-2507	`https://huggingface.co/Qwen/Qwen3-235B-A22B-Thinking-2507`

Computing infrastructure. All experiments were run on two servers, each equipped with eight NVIDIA A100 GPUs. The LLM checkpoints were deployed as inference services using vLLM, and both meta-agent and executor requests were served through these deployments.

Tool use and accounting. The tool registry is extensible and can be modified. Our experiments equip executor nodes with five such tools: a calculator, a code executor, a file reader, a file writer, and a test runner that reuses the code executor. Declaring a tool in the IR does not count as tool use, and our analyses count only observed invocations and their returned results.

Tool implementation. The calculator and code executor follow a text-convention and execution-feedback protocol. A node emits a fenced block, `calc` for the calculator and `exec` for the code executor, the runtime intercepts and runs it, and the returned result is appended as a new user message so the node can continue from real output. The calculator evaluates each expression with an abstract-syntax-tree evaluator restricted to a whitelist of arithmetic operators, math functions, and constants, and it never calls Python `eval`. The code executor, which also backs the test runner, concatenates the `exec` blocks into a single script and runs it in an isolated subprocess under `python -I`, inside a temporary working directory outside the repository, with a minimal environment that drops any secret-bearing variables, a wall-clock timeout, and truncated stdout and stderr fed back to the node. The file writer materializes each deliverable file from an `@<path>` line and its following code block into the run workspace, and the file reader surfaces the current workspace files in the node input. An audit hook further blocks reads of the held-out data directories, so an executed snippet cannot reach gold answers or hidden tests.

Trace logging for analysis. For analysis, we record the generated IR, compiled interaction order, node outputs, tool calls and results, terminal artifact, benchmark score, and token counts. These records support paired rescoring and topology analysis without additional model calls.

Table 6: Benchmark split sizes.

Benchmark	Total	Development	Test
AIME	60	30	30
MATH	5,000	500	4,500
HumanEval+	164	49	115
MBPP	500	150	350

B Benchmarks and Evaluation

We evaluate two mathematics and two code-generation benchmarks. AIME pools the 30 questions from each of the 2025 and 2026 contests. MATH uses the complete 5,000-problem test set, with a fixed 500-problem subset reserved as development set. HumanEval+ uses the 164 HumanEval problems with EvalPlus v0.1.10 base and additional tests. MBPP uses the 500 official test tasks, indexed 11–510, with the three published assertions retained as visible task specifications and as the pass@1 evaluator.

Per-benchmark scoring. AIME predictions are parsed from the final answer and compared as integers. MATH predictions are extracted from the final standalone `Answer:` field and graded using the symbolic-equivalence evaluator. HumanEval+ requires the generated program to pass both the EvalPlus base and plus tests; evaluation uses EvalPlus 0.3.1 on Python 3.10 with serial execution. MBPP requires the generated implementation to satisfy all the published assertions.

Information access and leakage control. The meta-agent and executor nodes receive the task statement and the applicable public contract. Hidden tests and canonical solutions are withheld. Offline trace analysis can use designs, predictions, execution events, and benchmark scores, but the resulting experience block contains only reusable, task-family-level principles. It excludes gold answers, hidden tests, task-specific equations, candidate solutions, and task IDs.

C Experience-Guided Design Evolution

Offline evolution procedure. Evolution runs between benchmark runs, not as an additional test-time loop. The initial task-family prompt is evaluated on the development data; score-labeled traces are summarized into recurring coordination successes and failures; and an additive experience block is distilled from these patterns. The selected block is then fixed for the reported execution. Model weights, the IR language, validator, compiler, executor, and scorer all remain unchanged. AIME and MATH use 30 and 500 development tasks, respectively; the MATH development set is 10% of the full 5,000-task benchmark. For HumanEval+ and MBPP we hold out a 30% development split. Table 6 summarizes the development-set sizes.

Prompt-template overview. Appendix I gives the fixed meta-agent message, its dynamic user and repair messages, the executor-node message, and the four retained task-family prompts. The templates preserve the operational fields and constraints used by the implementation; registry-generated text and task inputs are marked by placeholders. We also show the offline trace-analysis template.

Across families, the experience blocks emphasize four recurring principles: preserve exact task interfaces and candidate artifacts; add agents only for distinct marginal value; require objective evidence before overwriting a candidate; and keep the terminal node aligned with the benchmark's scoring format. Mathematics prompts additionally guard against ungrounded aggregation of scalar candidates, while coding prompts emphasize executable code continuity and boundary-condition checks.

D Failure Modes

For each failed execution, the trace audit identifies whether a correct or recoverable candidate ever appears and where the run first departs from that path. A *capability failure* occurs when no node

produces the correct deliverable. A *design failure* occurs when the system has a viable path but loses it through task interpretation, communication, verification, or terminal delivery.

Design-failure taxonomy. The qualitative audit uses four recurring modes, each tied to an observable trace signature. They compactly distill the broader findings: task distortion becomes specification drift; unverified repeated collaboration becomes error propagation; the design–execution gap becomes verification theater; and destructive aggregation and terminal artifact loss are grouped as post-solution degradation.

- **Specification drift.** An early node changes a task constraint, requested object, function interface, or return type; downstream nodes then solve the altered task consistently.
- **Error propagation.** A flawed plan or intermediate result is copied through later interactions without an independent derivation or objective check.
- **Verification theater.** A reviewer or tester declares success without executable evidence. A representative coding trace assigns a code executor to the reviewer but contains no tool-result event.
- **Post-solution degradation.** An intermediate node gives a correct candidate, but aggregation, revision, or terminal formatting overwrites or obscures it before scoring.

The verification mode is especially important: a declared reviewer, tester, or tool expresses design intent, but only an observed call and its returned result constitute executed verification. The audit therefore distinguishes declared architecture, executed behavior, and evidence-producing behavior. These semantic categories are used to organize representative traces rather than as exhaustive frequency estimates.

Cross-task trace evidence. On MATH, inspected regressions include task specifications that invent missing values or reverse the meaning of two functions. Reconciliation can also be destructive: in one trace both source paths return 5 but the reconciler introduces a different answer, and in another the correct $19/4$ is overwritten by an unsupported $39/8$. LiveCodeBench provides a combined error-propagation and verification-theater example: a planner and developer share an incorrect empty-stack assumption, after which a reviewer returns a pass verdict without executing the assigned tool.

Terminal artifact loss also appears across task families. In a separate LiveCodeBench prompt audit, 11 of 13 no-code terminal failures contain Python code upstream that is later replaced by JSON, a scalar, or review text. More generally, TIDE reduces no-code failures from 18 to 5 in the Qwen3-4B LiveCodeBench diagnostic while shifting other errors toward runnable but algorithmically wrong, timed-out, or runtime-failing programs.

Deterministic AIME audit. Because every AIME gold answer is an integer, we scan all node outputs in the 60-task initial and evolved runs. *Gold seen* counts wrong runs in which the gold integer appears somewhere in an intermediate output and is therefore a loose upper bound on design failure. *Strict loss* is the subset in which a node emits the gold integer in a standalone `Answer:` field but the terminal answer is wrong. *No gold* is the complementary proxy for capability failure.

Table 7: Failure decomposition on the 60 AIME tasks.

Prompt	Wrong	No gold	Gold seen	Strict loss
Initial	39	25	14	10
Evolved	29	26	3	1

Evolution reduces total errors from 39 to 29 and candidate-present failures from 14 to 3. Most notably, strict post-solution degradation falls from 10 cases to 1. Among the 29 remaining errors, 26 contain no observed correct candidate, so the residual failure mass is concentrated in executor capability rather than answer arbitration. The trace audit therefore isolates a concrete effect of evolution: it preserves correct intermediate answers through the terminal interaction. In this sense, trace-based evolution acts primarily as an orchestration regularizer: it teaches evidence gating and artifact preservation rather than increasing the executor's underlying solving capability.

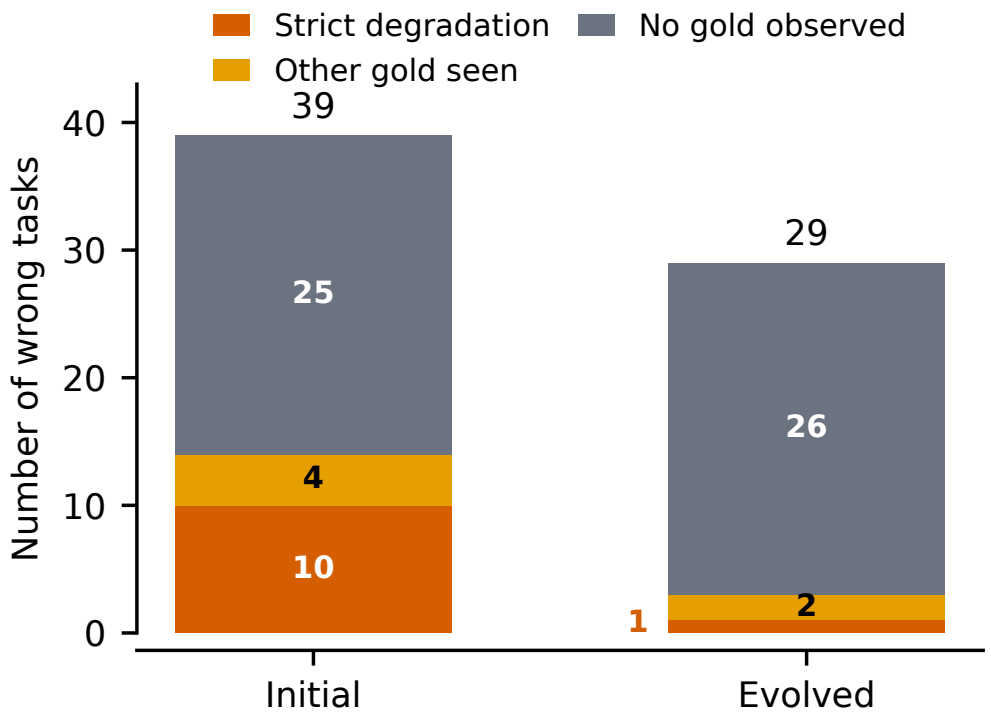


Figure 4: AIME failure decomposition before and after evolution.

E Additional Benchmark: Website Development

To probe portability beyond the four main benchmarks, we evaluate on the website subset of rSDE-Bench (Requirement-Oriented Software Development Benchmark). rSDE-Bench evaluates software-level coding from detailed functional requirements using automated requirement-level tests. Its website subset contains 45 realistic tasks and 541 test points, of which 293 are Basic and 248 are Advanced. Because each task requires a complete multi-file project, we drive both the meta-agent and the executor with a stronger code model, Qwen3-Coder-30B-A3B-Instruct, which every baseline also uses, and we run TIDE w/o Evolution with its base task-family prompt. We compare against a single-agent baseline, Vanilla, and three established multi-agent software-development systems, MetaGPT, ChatDev, and AutoGen. For TIDE w/o Evolution, the meta-agent emits a typed IR that the deterministic compiler builds and executes. For every baseline, an adapter recovers the generated project from the final output or workspace and normalizes it to a common directory layout, after which the same rSDE-Bench evaluator scores all methods.

Table 8: rSDE-Bench website-development results, in passed test points out of 293 Basic, 248 Advanced, and 541 total. All methods use the Qwen3-Coder-30B-A3B-Instruct model and the same evaluator; TIDE w/o Evolution uses the base task-family prompt. Best per column in **bold**.

Method	Basic (293)	Advanced (248)	Overall (541)	Acc. (%)
Vanilla	147	40	187	34.6
MetaGPT	45	0	45	8.3
ChatDev	209	71	280	51.8
AutoGen	166	56	222	41.0
TIDE w/o Evolution	**227**	**79**	**306**	**56.6**

Table 8 shows that TIDE attains the best score on both difficulty tiers and overall, at 56.6%, exceeding ChatDev, AutoGen, and Vanilla by 4.8, 15.6, and 22.0 points. The benchmark is demanding enough that a standard pipeline can collapse on it: MetaGPT solves no Advanced test point, whereas TIDE leads consistently across tiers. Two observations reinforce the paper's claims. First, this study drives both the meta-agent and the executor with a different, larger model than the main experiments, so the typed-IR design language and deterministic compiler transfer to new models, consistent with the executor-independence of the IR/compiler boundary. Second, the gain comes from the base engine alone: a meta-agent that expresses each system as a validated, compiled typed IR already outperforms hand-built agentic software pipelines before any experience-guided evolution.

Baselines. Vanilla is a single agent that emits the whole project in one response. ChatDev and MetaGPT use their standard role-based software-development MAS. AutoGen runs an Architect, an Engineer, and a Reviewer that take turns through `RoundRobinGroupChat` for up to six messages. An adapter then extracts the path-tagged code blocks from the final message into a standard project tree for scoring.

F ADDITIONAL ANALYSES

F.1 TOPOLOGY AND DIVERSITY ANALYSIS

We canonicalize each valid IR by its ordered interaction types, dependency pattern, and bucketed node counts. If p_s is the empirical frequency of skeleton s, the effective diversity is $D_{\text{eff}} = \exp(-\sum_s p_s \log p_s)$. Table 9 reports strictly paired Qwen3-4B-Instruct-2507 executor runs. Primitive columns are shares of interaction instances; fan-in is the fraction of designs with a multi-upstream interaction.

Table 9: Interaction-instance shares, effective topology diversity, and multi-upstream fan-in (%). "Other" combines fan-out, debate, vote, and conditional routing, whose individual shares are small in these runs. "MATH (dev)" denotes the 500-task development subset.

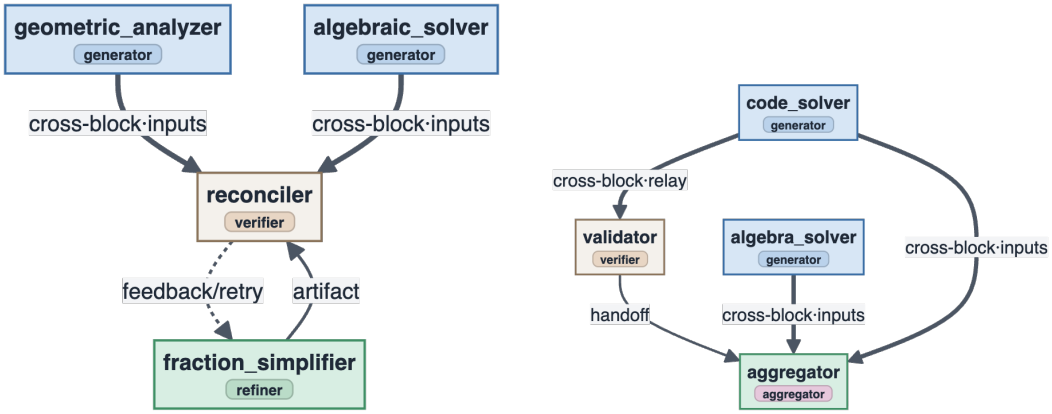
Benchmark	Prompt	Chain	Handoff	Refine	Other	D_{eff}	Fan-in
AIME	Initial	58.6	25.1	10.3	5.9	24.97	30.0
	Evolved	98.2	0.0	1.2	0.6	1.92	85.0
MATH (dev)	Initial	91.2	2.8	5.3	0.6	4.88	9.6
	Evolved	100.0	0.0	0.0	0.0	2.19	51.0
HumanEval+	Initial	17.0	38.8	43.6	0.6	8.41	0.0
	Evolved	19.6	39.8	40.6	0.0	10.97	0.0
MBPP	Initial	17.7	34.3	47.8	0.2	8.22	0.2
	Evolved	20.5	33.8	45.7	0.1	9.20	0.2

After evolution, mathematics and coding move in opposite directions. On math, designs converge to chain-based systems that merge several independent candidates: fan-in rises sharply, and D_{eff} falls from 24.97 to 1.92 on AIME. On coding, the policy instead keeps a broader mix of handoff and refinement, and D_{eff} rises from 8.41 to 10.97 on HumanEval+. These are correlated structural shifts rather than proof that any one topology alone causes the accuracy gain. Still, a single fixed template could not produce both patterns, so the guarded experience-guided evolution update appears to adapt the design prior to each task family.

F.2 CASE STUDIES OF GENERATED TOPOLOGIES

We inspect eight representative topologies emitted by the initial meta-agent on individual AIME problems. In each diagram, blue-bordered nodes are source entries, green-bordered nodes are terminals, warm neutrals mark intermediate arbitrators, and dashed edges denote iterative-refinement feedback. We name each topology by the coordination pattern it realizes.

Motif A: dual-path fan-in with gated reconciliation. Figure 5 shows a recurring skeleton already present before evolution: two independent generators feed a reconciler, after which a short refinement or aggregation step produces the terminal artifact. *Dual-View Reconciliation* pairs geometry and algebra and then locally refines the reconciled fraction. *Asymmetric Dual-Path Aggregation* keeps an explicit validator on one path and merges both streams in an aggregator, so disagreement is resolved before the sink. In both cases, several candidates are produced first, then merged under a dedicated arbitration node.

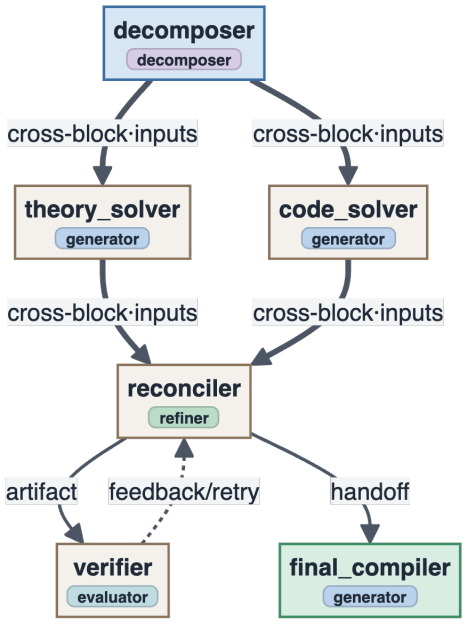


(a) Dual-View Reconciliation (b) Asymmetric Dual-Path Aggregation

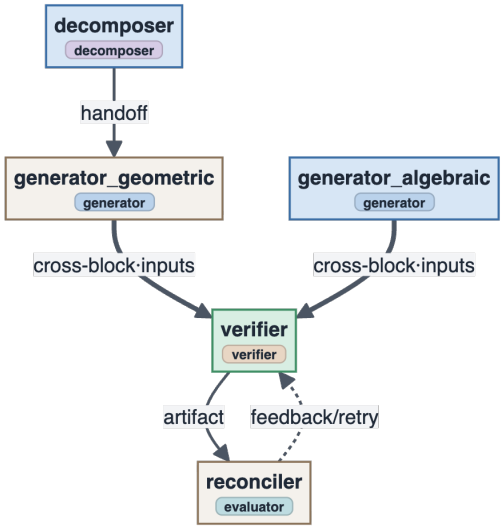
Figure 5: Dual-path fan-in topologies with local validation or refinement.

Motif B: decompose, then solve under cross-checks. Figures 6 and 7 retain an explicit decomposer, then route subproblems through complementary solvers. *Decompose, Dual-Solve, Compile* fans out to theory and code solvers, reconciles under a verify/refine loop, and finishes with a dedicated compiler node. *Dual-Generator Verify and Reconcile* and *Cascaded Math/Code Cross-Check* keep both a geometric (or symbolic) path and a code path, then force their products through a verifier/finalizer chain. *Reason-then-Enumerate Fan-in* lets a math reasoner dispatch a brute-force coder and then merges both streams. Relative to Motif A, these graphs spend more budget on structuring the problem before generation; the arbitration pattern at the sink remains the same evidence-preserving merge.

Motif C: specialized pipelines and search loops. Not every design is a pure two-path merge. Figure 8 shows two more specialized skeletons. *Verified Modular Pipeline* is nearly linear, but inserts a count-verification refine loop before the final modular step, so local checks do not rewrite the whole plan. *Geometry Multi-Path Reconcile* combines a geometry→algebra relay, an iterative candidate search, and a three-way fan-in into the reconciler. Together with Motifs A and B, these examples illustrate how the typed IR lets the initial meta-agent author problem-specific agents while reusing the useful set of interaction primitives.

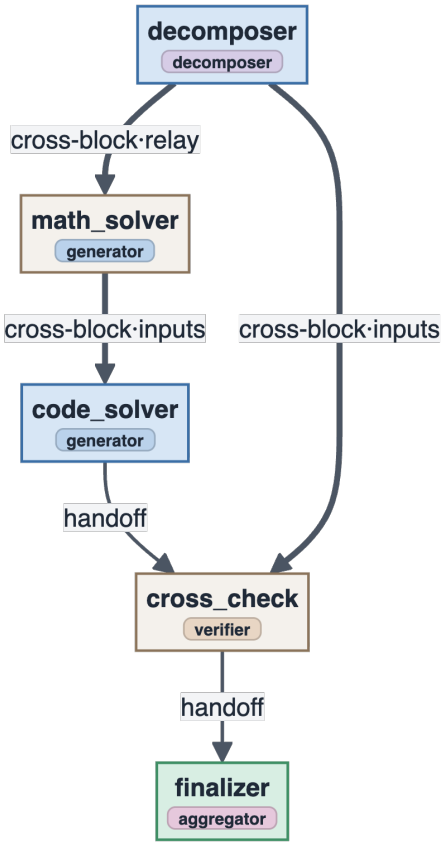


(a) Decompose, Dual-Solve, Compile

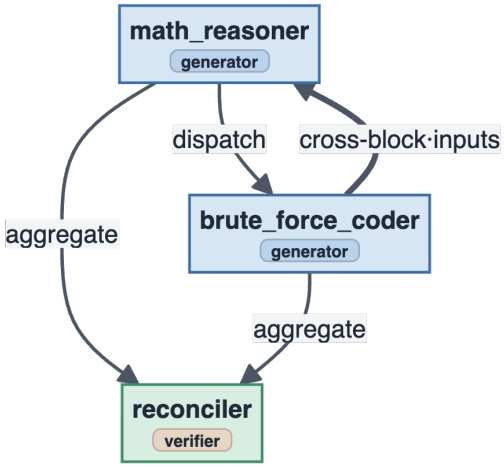


(b) Dual-Generator Verify and Reconcile

Figure 6: Decomposition-first topologies that fan out to complementary solvers.

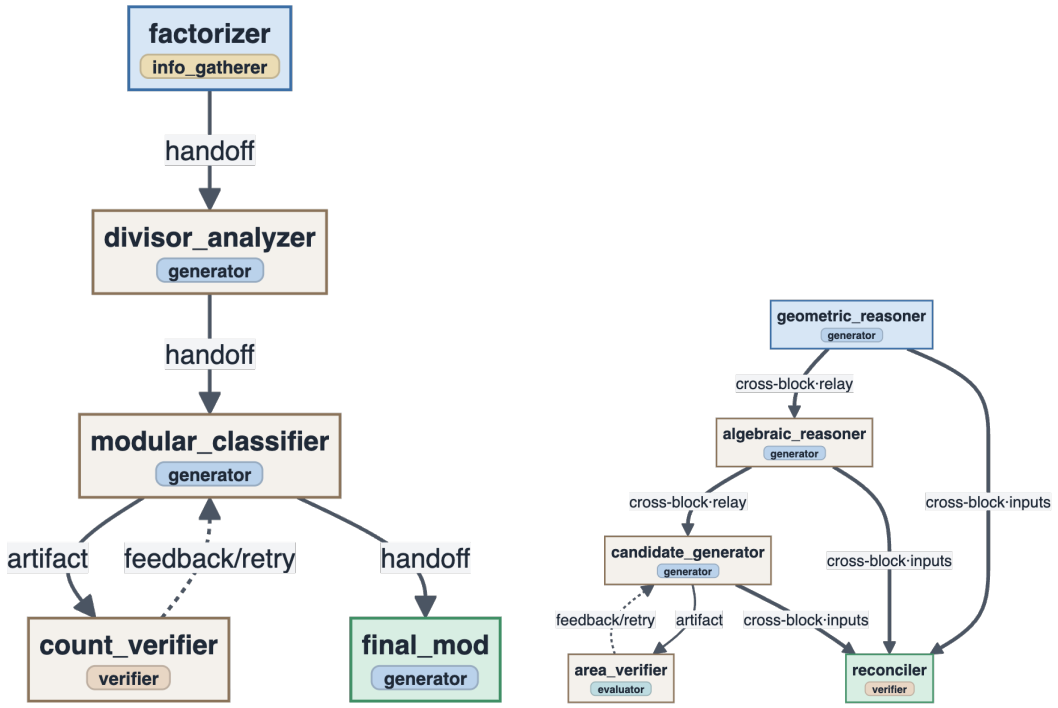


(a) Cascaded Math/Code Cross-Check



(b) Reason-then-Enumerate Fan-in

Figure 7: Cross-checked dual-generation designs with an explicit terminal merge.



(a) Verified Modular Pipeline (b) Geometry Multi-Path Reconcile

Figure 8: Specialized pipelines and search-oriented topologies.

G META-AGENT SCALING

To characterize how meta-agent capacity changes the synthesized systems, we fix the Qwen3-4B-Instruct-2507 executor, compiler, task-family prompts, and one-shot execution protocol. Each meta-agent independently samples one IR per task. The resulting statistics therefore compare checkpoint-level design policies.

Table 10: Aggregate structure across AIME, MBPP, and LiveCodeBench. "Nodes" is the mean over valid designs, "Single" is the percentage of one-node designs, and "Calls" is the total number of executor calls.

Meta-Agent	Nodes	Single	Calls
Qwen3-8B	2.32	36.4	2,249
Qwen3-32B	2.87	4.8	3,156
Qwen3-235B-A22B	2.18	0.5	2,287

Scale increases the propensity to use multi-agent coordination, but it does not monotonically increase system size or execution cost. The single-node rate falls from 36.4% at 8B to 4.8% at 32B and 0.5% at 235B. In contrast, both mean node count and total executor calls peak at 32B: 2.87 nodes and 3,156 calls, compared with 2.18 nodes and 2,287 calls at 235B. A larger meta-agent therefore does not simply add more agents or more executor computation.

The interaction mix instead reveals a shift in coordination policy. Chain, refinement, handoff, and other interactions account for 62.5/15.0/22.4/0.1% at 8B, 28.8/39.3/31.4/0.5% at 32B, and 23.8/71.8/4.3/0.0% at 235B. The 8B policy is primarily chain-based, the 32B policy uses a more balanced mixture, and the 235B policy strongly favors refinement. Thus, the 235B checkpoint is not reverting to direct single-agent execution; it prefers compact multi-node refinement structures. Together with the diminishing accuracy gains beyond 32B reported in the main paper, this supports the conclusion that a mid-sized meta-agent captures most of the scaling benefit, while further scaling mainly changes the structural prior.

H IMPLEMENTATION EXCERPTS

Codes 1 and 2 are simplified excerpts of the IR generation and compilation pipelines in `meta_agent/generator.py` and `compiler/compiler.py`. Boilerplate logging, error formatting, and auxiliary helpers are omitted for readability; function names, the validation-then-repair loop, and the deterministic lowering order are unchanged. Validation and compilation share the same primitive registry and dependency resolver, so a design accepted by the validator cannot acquire a different interaction signature during lowering. Node responses remain stochastic under sampling, but graph construction and scheduling are deterministic given the IR.

I PROMPT TEMPLATES

Prompt 1 shows the meta-agent system prompt. Prompt 2 shows the meta-agent user prompts. Prompt 3 shows the trace-analysis prompt, and Prompt 4 shows the executor-agent prompt. Prompts 5–8 show the evolved design experience for AIME, MATH, HumanEval+, and MBPP, respectively.

Code 1: IR Generation and Validation

```python
def generate_ir(
    *,
    llm,
    task_description,
    task_spec=None, strategy_block=None, design_directive=None,
    feedback=None, max_retries=5, base_model="unknown",
    design_id=None, postprocess=None,
) -> GenResult:
    last_error = None
    last_raw = None
    for attempt in range(max_retries + 1):
        system_prompt = build_system(strategy_block=strategy_block)
        user_prompt = build_user(
            task_description,
            feedback,
            task_spec=task_spec,
            design_directive=design_directive,
        )
        raw = llm.complete(system=system_prompt, user=user_prompt)
        last_raw = raw
        yaml_text = extract_yaml_block(strip_think(raw))
        if yaml_text is None:
            last_error = "No YAML block whose top level is mas_design."
            feedback = last_error
            continue
        obj, parse_error = safe_yaml_load(yaml_text)
        if obj is None:
            last_error = f"YAML parsing failed: {parse_error}"
            feedback = last_error
            continue
        if postprocess is not None:
            obj = postprocess(obj)
        resolved_id = design_id or f"{_slug(task_description)}_{uuid.uuid4().hex[:8]}"
        _inject_meta(
            obj,
            design_id=resolved_id,
            base_model=base_model,
            task_description=task_description,
        )
        ok, validation_error = validate_ir(obj)
        if not ok:
            last_error = validation_error
            feedback = f"IR validation failed: {validation_error}"
            continue
        return GenResult(
            ok=True,
            design=load_ir(obj),
            n_attempts=attempt + 1,
            raw=raw,
        )
    return GenResult(
        ok=False,
        error=last_error,
        n_attempts=max_retries + 1,
        raw=last_raw,
    )
```

Code 2: IR Compilation

```python
class Compiler:
    def compile(
        self,
        ir,
        *,
        workspace_subdir: Optional[str] = None,
        trace: Optional[ExecutionTrace] = None,
    ) -> CompiledMAS:
        # load_ir performs parsing and full semantic validation.
        design = load_ir(ir)
        subdir = workspace_subdir or design.meta.design_id
        workspace = Workspace(self.workspace_root / subdir)
        workspace.clear()
        role_by_id = {
            node.id: node.role
            for node in design.architecture.nodes
        }
        nodes = {
            node_id: build_node_fn(
                node_id,
                spec,
                llm=self.llm,
                workspace=workspace,
                role=role_by_id[node_id],
                max_tool_rounds=self.max_tool_rounds,
                exec_config=self.exec_config,
                trace=trace,
            )
            for node_id, spec in design.node_specs.items()
        }
        interactions = [
            compile_interaction(declaration, workspace=workspace)
            for declaration in design.architecture.interactions
        ]
        ordered_ids = [
            declaration.id
            for declaration in design.architecture.interactions
        ]
        dependencies = resolve_inputs(
            ordered_ids,
            {
                declaration.id: declaration.inputs
                for declaration in design.architecture.interactions
            },
        )
        terminal_ids = sinks(dependencies)
        return CompiledMAS(
            design=design,
            nodes=nodes,
            interactions=interactions,
            workspace=workspace,
            llm=self.llm,
            state_schema=construct_state_schema(design),
            interaction_order=topo_order(dependencies, ordered_ids)
    ,
            interaction_inputs=dependencies,
            sink_id=terminal_ids[0],
            trace=trace,
        )
```

Prompt 1: Meta-Agent System Prompt

Role. You are an expert designer of multi-agent systems (MAS). Given a task description, design one executable MAS by selecting functional roles, composing them with the registered interaction primitives, and specifying each node. You are the orchestrator and architecture designer, not the task solver. Return a typed YAML IR that a deterministic compiler can execute; do not draw an informal graph or answer the task yourself.

Registered interaction primitives.

- `handoff`: run one source node and transfer its artifact to one destination node; parameters are `from` and `to`.
- `sequential_chain`: run the nodes in the declared order; parameter `nodes` is a nonempty node list.
- `iterative_refinement`: alternate an `evaluator` and a `reviser` until a typed `exit_condition` holds or `max_rounds` is reached.
- `fan_out_merge`: run a `dispatcher`, parallel `workers`, and then a `merger`.
- `debate`: run parallel `participants` for a bounded number of `rounds`; a `moderator` returns the complete answer.
- `conditional_route`: a `router` selects a labeled branch, with an optional default branch.
- `vote`: run parallel `voters` and aggregate by `majority` or `merge`.

Registered roles and tools. Roles are `decomposer`, `generator`, `reviewer`, `evaluator`, `refiner`, `info_gatherer`, `aggregator`, `verifier`, `router`, and `tester`. Available tools are `calculator`, `code_executor`, `file_reader`, `file_writer`, and a test-runner contract implemented through the code executor. Allowed output formats are `code_files`, `json`, `markdown`, and `text`.

MAS Design Specifications.
`[GENERAL MAS DESIGN RULES]`

`[SPECIFICATION FOR GENERATING A VALID TYPED IR]`

Optional task-family experience. If supplied, append the block headed "Design experience" and treat it as reusable coordination guidance. It may not override the task or IR schema and must not contain task IDs, hidden tests, gold answers, instance-specific equations, or candidate solutions.

Required IR shape. The YAML object has the single top-level key `mas_design`. Its `meta` block contains a one-line task description; its `architecture` contains declared `nodes` and typed `interactions`; and `node_specs` maps every declared node ID to `persona`, `objective`, `instructions`, `output_format`, and `tools`.

Output. Return exactly one `yaml` fenced block whose top level contains only `mas_design`. Do not emit commentary before or after the block.

Prompt 2: Meta-Agent User Prompts

Initial user message.
`[TASK-FAMILY DELIVERY CONTRACT]`

Task description:
`[COMPLETE CURRENT TASK]`

`[OPTIONAL DESIGN DIRECTIVE]`

Output the MAS design IR for this task. Return exactly one complete `yaml` fenced block and no other text.

Task-family delivery contract. This block states what the terminal artifact must contain, how it will be scored, which public task information is available, and the exact final format. It changes across different benchmarks; the fixed system prompt and IR language do not.
Optional design directive. This field may impose a run-level structural constraint, such as a maximum node count. It cannot add facts to the task or alter the benchmark scorer.
Repair variant. If the response lacks a YAML block, cannot be parsed, or fails typed IR validation, resend the same system and task-family messages and append exactly one of the following diagnostics:

- "No YAML block was found. Return only one `yaml` block whose top level is `mas_design`."
- "YAML parsing failed: `[PARSER DIAGNOSTIC]`. Correct the syntax and return the complete IR again."
- "IR validation failed: `[SCHEMA OR GRAPH DIAGNOSTIC]`. Correct the violation and return a complete replacement IR."

The diagnostic may report missing fields, unknown primitives, unresolved node references, invalid round bounds, cycles, unreachable nodes, or a non-unique terminal interaction. It never includes a gold answer, evaluator score, hidden test, or task-correctness feedback.
Minimal output skeleton.

```
mas_design:
  meta:
    task_description:  [ONE-LINE DESCRIPTION]
  architecture:
    nodes:  [{id:  solver, role:  generator}]
    interactions:
      - {id:  solve, type:  sequential_chain,
         params:  {nodes:  [solver]}}
  node_specs:
    solver:
      persona:  [FUNCTIONAL PERSONA]
      objective:  [ABSTRACT OBJECTIVE]
      instructions:  [[METHOD-LEVEL INSTRUCTION]]
      output_format:  markdown
      tools:  [none]
```

Prompt 3: Trace Analysis Prompt

Role. You are an offline analyst of multi-agent system designs. Contrast successful and failed development executions and distill reusable task-family design experience. You are not solving any individual task, and your output will be used only to revise the meta-agent's task-family prompt.
Inputs.

- Current task-family contract: `[CURRENT TASK-FAMILY CONTRACT]`.
- Development records: `[TASK STATEMENT, GENERATED IR, COMPILED INTERACTION ORDER, NODE ROLES, TOOL EVENTS, TERMINAL PREDICTION, BENCHMARK SCORE]`.
- Aggregate summaries: `[ACCURACY BY TOPOLOGY, NODE COUNT, INTERACTION TYPE, TOOL OUTCOME, AND TERMINAL FAILURE TYPE]`.

Analysis procedure.

- Compare recurring mechanisms rather than isolated examples. Identify which additions supply independent evidence and which merely duplicate the same model behavior.
- Separate knowledge errors from coordination errors. Coordination errors include destructive overwriting, malformed terminal artifacts, interface drift, unconsumed tool output, correlated "independent" agents, and aggregation without evidence.
- Inspect whether the terminal interaction preserves the best available candidate and the benchmark's required output format.
- For tool-using designs, distinguish declaring a tool from successfully invoking it and consuming the returned result.
- Prefer the smallest intervention supported by repeated trace evidence. Do not prescribe a complex topology merely because one successful example used it.

Information boundary. Never reproduce or encode a gold answer, hidden test, task ID, instance-specific equation, substituted value, full candidate solution, or a rule that identifies one development instance. Task text may be inspected to classify the failure, but the extracted advice must apply to new tasks in the same family. If a proposed rule cannot be stated without instance-specific content, discard it.
Required output.

- **Recurring successful mechanisms:** concise evidence-backed patterns and the conditions under which they helped.
- **Recurring failure mechanisms:** the affected topology or interface, its observable trace signature, and why it harmed the terminal result.
- **Candidate experience block:** at most eight imperative bullets governing node marginal value, topology choice, disagreement arbitration, tool evidence, and terminal-format preservation.
- **Leakage check:** return `PASS` only if the candidate block contains no prohibited instance information; otherwise return `FAIL` and remove the offending content.

Do not claim that the candidate improved performance. A separate paired development run evaluates the candidate, and the external selection rule keeps it only when measured accuracy does not decrease.

Prompt 4: Executor Agent Prompt

System message assembled for each executor node.

Your role: `[PERSONA FROM NODE SPEC]`
Your objective: `[OBJECTIVE FROM NODE SPEC]`
Instructions:

- `[INSTRUCTION 1 FROM NODE SPEC]`
- `[INSTRUCTION 2 FROM NODE SPEC]`
- `[ADDITIONAL INSTRUCTIONS, IF ANY]`

Output contract:

- For `markdown`, give a clear Markdown response.
- For `text`, give concise plain text.
- For `json`, return one parseable JSON object.
- For `code_files`, emit each complete file as one `@relative/path` line followed by one code block.

Tool contract, included only for assigned tools.

- Calculator calls contain one pure Python arithmetic expression per `calc` line. Use `**` for powers; do not include assignments, units, LaTeX, equality statements, or explanations.
- Code execution calls contain a complete, independently runnable Python program in an `exec` block. Print every result that must be read.

Optional structured control contract. For an evaluator, router, or moderator, the executor appends a typed directive and required JSON keys. Examples include a Boolean pass field with feedback, a labeled route, or a complete `final_answer`. These fields are parsed by the scheduler and are not authored by the meta-agent.

User message.
`[ORIGINAL TASK AND UPSTREAM INTERACTION ARTIFACTS]`

`[OPTIONAL REMINDER OF THE STRUCTURED CONTROL CONTRACT]`

Tool-result follow-up, when applicable.
Your previous response was: `[NODE RESPONSE CONTAINING TOOL CALLS]`.
The tool returned: `[EXECUTION OR CALCULATION RESULT]`.
Continue from the actual result and provide a complete final response.

Prompt 5: AIME Evolved Design Experience

Selected evolved task-family design contract. It is prepended to the current AIME problem in the meta-agent user message.

- The deliverable is the final integer requested by the problem. The terminal artifact must preserve the complete derivation and end with a standalone line in the exact form `Answer: [integer]`.
- All executor roles use the same small model; a role name does not automatically provide stronger judgment. Prefer a compact design, normally one main solver. Add a second solver and an adjudicator only when the second method is genuinely different and independently checkable. Use at most three nodes and three interactions.
- It is not recommended to use `fan_out_merge`. Its dispatcher artifact is passed to every worker; for scalar mathematics this can make workers repeat the same total and cause a merger to add it incorrectly.
- If two solving paths are used, make them independent sources (the second path explicitly uses `inputs: []`) and require substantively different methods. The terminal adjudicator first extracts each path's `Answer`. If they agree, copy that integer unchanged. If they disagree, adjudicate only by re-deriving a key equality or objectively recomputing it with `calc`/`exec`. Never sum, average, concatenate, or invent a third candidate. Preserve the main solution unless objective evidence refutes it.
- An evaluator/verifier checks explicit equalities, boundaries, enumeration completeness, and arithmetic; it does not re-solve the whole task from subjective confidence. Use `iterative_refinement` only when an executable check exists, with `max_rounds: 1`.
- The meta-agent chooses abstract methods only. Do not place problem-specific equations, case splits, substituted values, intermediate results, or candidate answers in any node specification. Executor nodes must derive all task-specific content from the original problem.
- Assign `calculator` or `code_executor` for exact arithmetic or finite enumeration and instruct the node to issue a real `calc`/`exec` call. Do not add a nominal verifier when no tool or distinct evidence is needed.

Prompt 6: MATH Evolved Design Experience

Selected evolved task-family design contract. It is prepended to the current MATH problem in the meta-agent user message.

- The deliverable is the mathematical object explicitly requested. If auxiliary variables are introduced, convert them back to the requested quantity in the final step.
- Every executor role uses the same small model; role labels do not create stronger judgment. Even for a difficult ordinary problem, default to exactly one `generator` node and one `sequential_chain`, with that solver's complete response as the terminal artifact. Do not append a verifier merely to increase confidence.
- Use at most three nodes and three interactions, and use only `sequential_chain`. Do not use `handoff`, `fan_out_merge`, `debate`, `vote`, or `iterative_refinement`; these structures can cause same-model roles to repeat, rewrite, or overwrite a correct scalar or symbolic answer.
- Use a second structure only when two naturally independent methods exist, such as symbolic derivation and executable finite enumeration. Create two independent generator sources and one reconciler; the second source must explicitly use `inputs: []` and must not see or continue the first answer.
- The reconciler first extracts the two final answers. If they are equivalent, preserve the requested representation. If they differ, adjudicate only through `calc`/`exec` or a fresh derivation of the critical equality. Never sum, average, concatenate, or invent a third candidate; retain the main solution unless objective evidence refutes it.
- The terminal artifact must retain a complete solution and end with a standalone `Answer: $[final answer]$` line. Do not output literal angle brackets. A verification JSON object, pass field, feedback message, or review alone is not a valid terminal artifact.
- Preserve the exact object and representation requested by the task: tuple order, set/interval endpoints, matrix shape, the complex unit, angles, textual labels, and numeral bases must not be rewritten casually.
- The meta-agent describes abstract methods only and must not prewrite task-specific equations, roots, substituted values, intermediate results, or candidate answers in node specifications.
- Assign and actually invoke a calculation or execution tool only for exact arithmetic or finite enumeration. Do not add nodes when no executable, independent check is available.

Prompt 7: HumanEval+ Evolved Design Experience

Selected evolved task-family design experience. The first four bullets are the base delivery contract; the remaining bullets are the additive experience block distilled from the designated development traces.

- The deliverable is one complete, runnable Python implementation, evaluated by the original HumanEval hidden tests and the additional EvalPlus tests.
- Preserve the function name, signature, and return contract in the problem. Do not rewrite the interface or hard-code only the documented examples.
- The implementation must cover the general inputs and boundary cases represented by the specification. Intermediate nodes may analyze, implement, test, and review.
- The final node must directly output complete code containing the original function signature and all necessary imports, inside exactly one `python` code block. A plan, test verdict, or review is not a valid terminal artifact.
- Before implementation, turn the natural-language requirements into distinguishable behavioral invariants and construct boundary examples that separate nearby interpretations. Check quantifier scope, order and extrema, boundary conventions, overlapping conditions, and return types.
- Verification must trace the current code's control flow, aggregation logic, and every return branch against those invariants. Unexecuted self-written examples, role labels, or subjective pass statements are not evidence.
- Allocate collaboration according to unresolved uncertainty. Every additional role must contribute a different implementation or verification value; it must not rewrite a working implementation without evidence. Replace the current code only when stronger evidence supports the revision.
- If a tool is declared, execute minimal tests targeting ambiguity and boundaries and consume the real result. Without execution, perform a concrete case-by-case trace. Every handoff and the terminal step must preserve the complete executable code as the primary artifact.

Prompt 8: MBPP Evolved Design Experience

Selected evolved task-family design experience. The first four bullets are the base delivery contract; the remaining bullets are the additive experience block distilled from the designated development traces.

- The deliverable is complete, runnable Python code implementing the requested function. The three assertions in the task are public behavioral specifications; all three must pass.
- The final node must directly output the complete implementation inside exactly one `python` code block. A prose explanation, test verdict, or example value is not a valid terminal artifact.
- The implementation must satisfy the general task description and may not hard-code the concrete inputs from the public assertions.
- Intermediate nodes may analyze boundaries, implement, test, and review, but the terminal interaction must preserve directly executable code.
- Treat the original task as the sole factual source. Every handoff must preserve the exact public interface and behavioral invariants, including identifier spelling and case, parameters, return form, and assertion semantics. Do not rewrite the interface while paraphrasing it.
- Allocate collaboration according to unresolved uncertainty. Every additional role or interaction must provide a distinct analysis, implementation, or verification contribution. Avoid downstream rewrites that introduce new contract drift without evidence.
- Verification must compare the current code with every public specification concretely. Declaring a tester or tool is not evidence. If a tool is used, consume the real result; otherwise inspect the interface, assertions, and boundary semantics item by item and return any discovered defect to code revision.
- Preserve executable code as the primary artifact across analysis, implementation, and review. Replace it only with a revision supported by stronger evidence, and terminate with the complete code rather than a pass statement, score, or commentary.