THINKING BEFORE THINKING: SCALING AGENTIC INFERENCE THROUGH META-REASONING

Anonymous authors
Paper under double-blind review

ABSTRACT

As agents take on longer and more complex problems, controlling the execution becomes a task in its own right. Each step in the run brings new control choices, like which partial work to build on, whether to start fresh, or when to stop. We introduce *agentic meta-reasoning*, an inference-time harness that makes these choices an explicit and structured reasoning process. Workers carry out the task-level computation, while a controller consolidates what the run has established, explores next options, assesses what each option is worth under the remaining budget, and dispatches the chosen work with context drawn from persistent memory. Between decisions the controller carries only a compact account of the run rather than replaying its full history. Our baselines span production coding agents and research harnesses, together with a Direct Control Agent using the same workers and compute budget allowance. On ProgramBench, which tests long-horizon agentic capability through program reconstruction, meta-reasoning achieves 71.5% with GPT-5.5 against 58.0% for Codex; with Opus 4.8 it achieves 67.2% against 65.5% for Claude Code. On the other benchmarks, spanning abstract reasoning, multi-domain long-horizon reasoning, and proof generation, it gains between 3.6 and 4.2 points over direct control, averaged across three frontier models. It keeps improving over the tested budget ranges where direct control plateaus, though its overhead can hurt at small budgets. Artifact-graph analysis reveals more reuse of earlier work, higher coverage of correct solutions in most settings, and nonuniform gains in final selection. These results indicate that spending computation on structured control becomes more important as agents scale to longer runs.

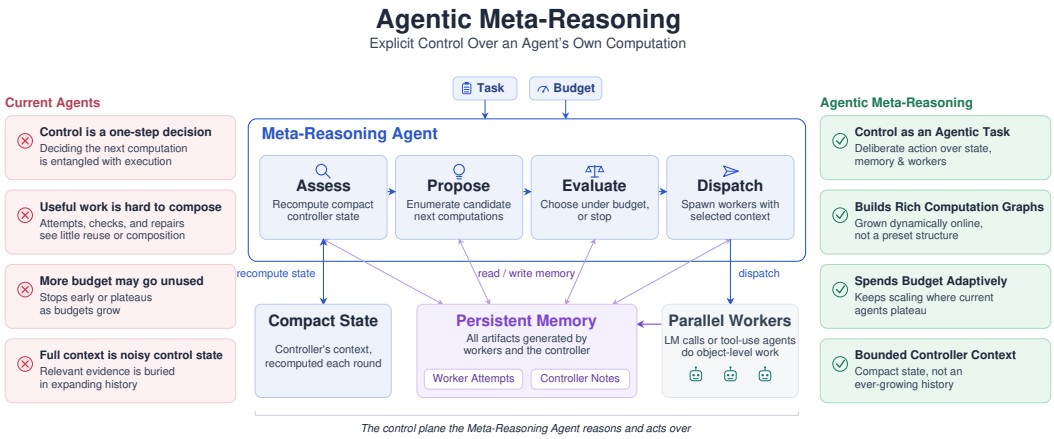

Figure 1: **Agentic meta-reasoning.** A controller assesses progress, proposes candidate computations, evaluates them under the remaining budget, and dispatches workers with selected context from persistent memory. Worker outputs become new artifacts, and the selected contexts induce an artifact graph that grows throughout the run. Between cycles, the controller carries a compact state rather than replaying the full history.

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.

1 INTRODUCTION

Language model agents increasingly spend a large number of model calls on a single problem, and a considerable share of those calls are decisions about what to do next. Consider a math agent working on a proof, holding a candidate whose main argument looks sound but whose central lemma is unverified. It can attempt the lemma, ask a second worker to check the argument it already has, or abandon the approach for a different one. Each option costs model calls, and those spent on the wrong one are wasted. For an agent working autonomously toward a goal, this choice is part of solving the problem, and a capability distinct from producing the next step of the proof.

This is a problem of *metacognitive control*: assessing one's own progress and using that assessment to decide what to do next (De Sabbata et al., 2024; Liu et al., 2026). In a single model call, the object of that control is the length of a chain of thought. In an agent it is the work the run has already produced, so it must decide which results to trust, what to build on, and when to stop (Li et al., 2025; Xiang et al., 2026). A mistaken judgment costs more than the calls it consumes. It can keep a failed approach alive for the rest of the run, or throw away a correct answer the agent has already found (Brown et al., 2024). Existing agents often interleave these decisions with the object-level work, taking each control decision in a single step conditioned on their accumulating history (Yao et al., 2023b; Shinn et al., 2023; Shen et al., 2023; Packer et al., 2023; Zhang et al., 2025a). In this paper we argue that these decisions deserve an agentic reasoning process of their own. An agent can deliberate and inquire before it commits, reading back an earlier result or asking a worker to check one. That deliberation has a structure of its own, consolidating what the run has established, exploring what could be done next, and assessing what each option is worth.

We call this approach *agentic meta-reasoning*: the agent reasons about and acts on its own inference process. We implement it as a new inference-time *harness*, the runtime that coordinates model calls and manages the run. Our Meta-Reasoning Agent separates the workers, which do the task-level work, from the controller, which decides what work to assign. The controller keeps a compact assessment of progress. Full worker outputs remain in persistent memory, where they can be retrieved when needed.

Each control cycle runs four stages: the controller assesses what has changed, proposes next computations, evaluates their value under the remaining budget, and dispatches the chosen work along with the earlier outputs each worker should see. Each of these stages is an agentic process in its own right, with its own instructions and tools. The controller model calls are charged against the same budget as the workers', so the extra deliberation has to earn what it costs. We record the resulting work as an *artifact graph*. Every stored output is an artifact, and an edge marks one artifact being supplied as context for producing another. The graph shows where a run branches and which later steps build on earlier ones, letting us diagnose it by the work it produced rather than by its final score alone.

We evaluate the harness on IMO ProofBench-Advanced, ARC-AGI-2, LongCoT-mini, and ProgramBench, spanning proof search, abstract reasoning, long-horizon reasoning, and program reconstruction, using Gemini 3.1 Pro, GPT-5.5, and Opus 4.8. A worker is either a single model call or a coding agent that uses tools. Our baselines span production coding agents and research harnesses, together with a Direct Control Agent that holds workers and actions fixed so that only the control differs, choosing each action in a single call over accumulated history.

At the main allowances of 100 model calls on the reasoning benchmarks and 1200 on ProgramBench, meta-reasoning yields higher point estimates in all 12 matched comparisons, with mean gains of 3.6–4.2 score points by benchmark. On ProgramBench with GPT-5.5, mean per-problem test-pass rate rises to 71.5% as compared to 58.0% for Codex and 63.7% for direct control. The performance scales as allowance grows where direct control plateaus, and the artifact graphs show more reuse of earlier work and correct candidate present in more runs.

2 AGENTIC META-REASONING

Agentic meta-reasoning treats the choice of what to compute next as its own agentic task, separate from the object-level computation it directs. We describe the memory and action interfaces first, then the control cycle and its computational cost.

2.1 MEMORY, STATE, AND WORKER CONTEXT

An *artifact* is a stored output from a worker or the controller: for example, a proof attempt, a critique, or a note recording an unresolved gap. Each artifact has a stable identifier and can be retrieved later. For a task x, let M_t denote artifact memory during control cycle t; this notation suppresses updates made within a cycle. Rather than carrying the full artifact contents between cycles, the controller maintains a compact, self-authored textual state s_t describing what the run has established and what remains uncertain. This state is rewritten each cycle; the underlying artifacts persist in memory. The controller also sees an index of artifact identifiers and titles and can retrieve full contents on demand. Thus, earlier work remains accessible without being replayed in every control prompt.

A worker receives the task x, a controller-authored instruction g, and a selected set of prior artifacts $C \subseteq M_t$. We denote the resulting output by $y = W(x, g, C)$. The instruction specifies *what* to do; the selected artifacts specify *what prior work* to build on. For example, a worker checking an unverified lemma might receive the existing proof and a critique identifying the gap, whereas an independent attempt could receive no prior artifacts. Workers do not see the controller's private state or deliberation. The worker interface covers both single-call model workers and multi-turn coding agents. For the latter, W is shorthand for a tool-using execution that may inspect or change an external environment; the explicit inputs above are not a complete causal description of its output.

2.2 ACTIONS AND THE ARTIFACT GRAPH

The controller can inspect stored work, record its own notes, launch workers, or stop by submitting an artifact. For a collection of artifact identifiers I, $\mathrm{Read}(I)$ retrieves the corresponding artifacts, and for new content u, $\mathrm{Write}(u)$ stores it as an artifact with a fresh identifier; these are epistemic actions, in which the controller leaves notes for itself to organize what it knows before acting on the task (Kirsh & Maglio, 1994). To advance the object-level work it launches a batch of k parallel workers, supplying each with an instruction g_i and artifact context C_i, so that $\mathrm{RunWorkers}(\{(g_i, C_i)\}_{i=1}^k) = \{W(x, g_i, C_i)\}_{i=1}^k$. Returned artifacts receive fresh identifiers and are stored together with the identifiers of their inputs, and the same interface supports a fresh attempt, a targeted repair, or a synthesis of earlier results, so there is no fixed schedule of worker roles. The run ends with $\mathrm{Stop}(y^\star)$, where $y^\star$ is an artifact already in memory; to submit a fresh synthesis the controller must first have a worker produce one.

Every returned worker output is stored with the identifiers of the artifacts provided as its context. These recorded dependencies form an *artifact graph* $G = (V, E)$, with artifacts as nodes and

$$E = \{(u, v) \in V \times V : v \text{ is a worker output and } u \in C_v\}. \tag{1}$$

Because a worker can only receive artifacts that already exist, the graph is directed and acyclic. Its structure is determined online: a fresh attempt starts a new branch; a repair extends an earlier result; a synthesis can combine multiple branches. An edge means that an artifact was *supplied* to a worker, not that the worker used it successfully. The graph also omits unrecorded dependencies through external tools or shared environment state. In Section 3, we examine the worker-artifact subgraph to characterize the computation that each run produced.

2.3 A CONTROL CYCLE

Each control cycle separates four questions: *What have we learned? What could we do? Which option is worth its cost? What should the worker receive?* The corresponding stages—*Assess*, *Propose*, *Evaluate*, and *Dispatch*—run sequentially. Each begins with a fresh context and has its own instructions and permitted memory operations; a stage may use multiple model calls to inspect artifacts or deliberate. The notation below shows their principal inputs and outputs. Memory and budget can change within a cycle as stages read, write, and consume model calls.

Assess: What have we learned? Given the previous state s_{t-1} and new worker artifacts ΔM_t, Assess updates the controller's account of progress:

$$s_t = \mathrm{Assess}(x, s_{t-1}, \Delta M_t; M_t). \tag{2}$$

The semicolon denotes access to persistent memory rather than inclusion of all its contents in the prompt. Assess can retrieve older artifacts to check its interpretation of new results. In the proof example, it might record that the main argument remains promising but its central lemma is still

unverified. The compact state is a working assessment, not an authoritative summary: if it overlooks an error or a useful candidate, subsequent control decisions can suffer.

Propose: What could we compute next? From the updated state, Propose generates candidate next computations, including the option to stop:

$$\mathcal{A}_t = \mathrm{Propose}(x, s_t; M_t). \tag{3}$$

For an unverified lemma, it might propose an independent check, a targeted proof attempt, or a different strategy. Propose is not given the remaining budget explicitly. This separates generating alternatives from judging their affordability, so potentially valuable but expensive options can enter the candidate set.

Evaluate: Which option is worth its cost? Evaluate considers the proposed actions and the budget remaining *after* the calls spent on earlier stages:

$$\tilde{a}_t = \mathrm{Evaluate}(x, s_t, b_t^{\mathrm{eval}}, \mathcal{A}_t; M_t), \qquad \tilde{a}_t \in \mathcal{A}_t. \tag{4}$$

It may prefer checking the lemma over restarting the entire proof, or stop if the expected benefit of further work appears too small. This is a prompted, qualitative assessment of computational value, not an exact optimization or a learned value-of-computation estimator.

Dispatch: What should the worker receive? Dispatch turns the selected proposal into an executable action:

$$a_t = \mathrm{Dispatch}(x, s_t, \tilde{a}_t; M_t). \tag{5}$$

If more work is selected, it writes the worker instructions, chooses their artifact contexts, and launches the workers. If stopping is selected, it submits an existing artifact. Returned outputs are stored and become available to the next Assess stage. Crucially, choosing a computation includes choosing its inputs: the controller can send a worker the precise proof and critique needed for verification without exposing unrelated attempts or its own deliberation.

2.4 Direct Control and the Cost of Deliberation

The *Meta-Reasoning Agent* uses the four-stage cycle above. The matched *Direct Control Agent* has the same worker models and interfaces for launching workers, selecting artifact context, writing memory, and stopping, but chooses its next action in a single controller call over accumulated history and the remaining budget. It has neither the separately rewritten compact state nor the four-stage deliberation process. The comparison thus isolates the *combined control design*, but it does not separately identify the effects of staging, state compression, or memory access.

Both agents receive a nominal budget of B model calls. Controller reasoning is charged just like worker computation: if cycle t consumes c_t controller calls and w_t worker calls, the remaining allowance is

$$b_{t+1} = b_t - c_t - w_t, \qquad b_0 = B. \tag{6}$$

A call spent deciding what to do is therefore a call unavailable for doing it. Budget enforcement occurs between control cycles, so a dispatched batch can finish after the nominal allowance is reached; if the agent does not stop before exhaustion, the harness forces a submission. This accounting makes the central tradeoff explicit: structured control is useful only when its decisions improve the work performed enough to repay their overhead.

3 Diagnosing Agentic Inference Through Artifact Graphs

Final accuracy tells us whether an agent succeeded, but not where its computation succeeded or failed. One run may never produce a correct proof; another may produce one and then submit a flawed revision. The artifact graph lets us distinguish these cases wherever intermediate solutions can be graded. We analyze how a run spends its allowance, what dependencies it creates between worker outputs, whether it finds and submits a correct candidate, and how well it judges the candidates it produces. Appendix D gives the full metric definitions.

Computation and recorded dependencies. Because an agent can stop early, we measure budget utilization as $U = (N_{\mathrm{ctrl}} + N_{\mathrm{work}})/B$, where B is the nominal model-call allowance. We then

examine the *worker-artifact graph*, whose nodes are worker outputs and whose edges indicate that one output was supplied as context to a later worker. Node and edge counts measure the amount of work and its dependencies; fan-in, longest-path depth, and maximum width distinguish independent attempts from chains of revision and the combination of earlier results. We analyze controller-authored notes and explicit memory reads and writes separately, together with the size of the state carried between control cycles.

Finding a correct answer versus submitting one. Let V_{sol} contain the artifacts that can be graded as candidate solutions, including the submitted artifact $y^\star$, and let $\ell(y) \in \{0, 1\}$ indicate correctness. Write C for the event that a run contains at least one correct candidate and S for the event that its submission is correct. Since $S \subseteq C$,

$$\Pr(S) = \Pr(C)\ \Pr(S \mid C). \tag{7}$$

We call $\Pr(C)$ *coverage*: the accuracy an oracle selector could achieve from the candidates the run actually produced. We call $\Pr(S \mid C)$ *selection*: the probability of submitting a correct candidate, conditional on having found one. A failed run is *coverage-bound* if it never produced a correct candidate, and *selection-bound* if it produced one but submitted an incorrect one. Coverage can also be measured at intermediate call checkpoints to show *when* correct candidates first become available. This decomposition uses binary correctness; on IMO ProofBench-Advanced it is distinct from the benchmark's partial-credit score.

Recognizing correct work. Workers report confidence in their outputs, and the meta-reasoning controller's Assess stage records verdicts on candidates it reviews. We measure how well each signal distinguishes correct from incorrect candidates using Type-2 AUC: the probability that a correct candidate receives a higher score than an incorrect one, with half credit for ties (Galvin et al., 2003; Fleming & Lau, 2014). This measures *discrimination*, not whether confidence values are calibrated. Note that Direct control has no corresponding explicit controller-verdict signal as it has no isolated assessment stage.

Choosing among final candidates. To examine selection beyond coverage, we identify the *convergence frontier*: the deepest terminal worker artifacts in a run's recorded dependency graph. Where these artifacts have correctness labels, let q_F be the fraction that are correct. We define *frontier selection gain* as

$$\mathrm{FSG} = \mathbb{E}[\ell(y^\star) - q_F \mid C]\,, \tag{8}$$

comparing the actual submission with a uniform random choice from that run's frontier, among runs containing a correct candidate. A submitted answer need not lie on the frontier: an earlier, shallower artifact may be better. We therefore interpret this measure alongside frontier size and the frequency of frontier submissions.

4 Experimental Results

We evaluate on four benchmarks: IMO ProofBench-Advanced (Luong et al., 2025) (30 proof problems), ARC-AGI-2 (Chollet et al., 2026) (120 abstract visual reasoning tasks), LongCoT-mini (Motwani et al., 2026) (507 long-horizon reasoning problems), and ProgramBench (Yang et al., 2026) (200 program reconstruction tasks). We refer to the first three as the reasoning benchmarks. We evaluate Gemini 3.1 Pro, GPT-5.5, and Opus 4.8, accessed through their respective APIs, with nominal budgets of 100 model calls per problem on the reasoning benchmarks and 1200 on ProgramBench. Every controller and worker call is charged against the same allowance. Our primary baseline is a matched Direct Control Agent with the same workers and action interfaces. We also compare against RLM (Zhang et al., 2025a) on the reasoning benchmarks and mini-SWE Agent (Yang et al., 2024), Claude Code (Anthropic, 2026), and Codex (OpenAI, 2025) on ProgramBench. Full experimental details are given in Appendix B.

4.1 Meta-Reasoning Improves Performance Across Tasks and Models

Does explicit control improve performance with the same workers? At the main budgets, meta-reasoning achieves higher scores in all 12 matched model–benchmark comparisons (Appendix Table 1). Averaged across the three models, the gains over Direct Control are 4.0 points on IMO ProofBench-Advanced, 4.2 on ARC-AGI-2, 3.6 on LongCoT-mini, and 3.8 on ProgramBench. The

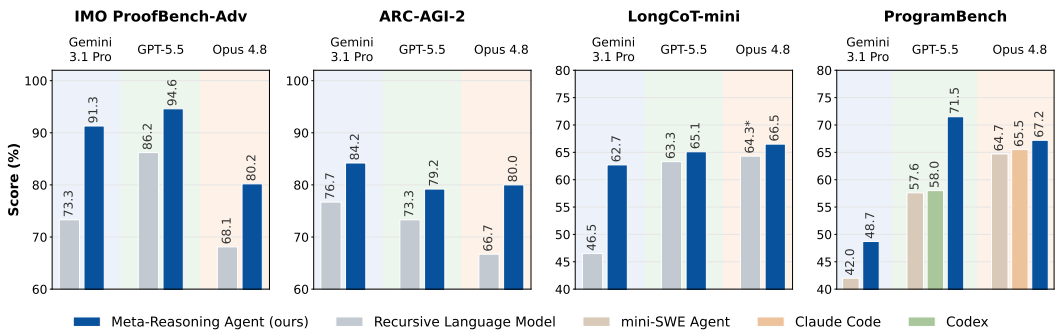

Figure 2: **Performance across four benchmarks and three models.** The Meta-Reasoning Agent is compared against external research harnesses and production coding agents. Nominal budgets are 100 model calls on the reasoning benchmarks and 1200 on ProgramBench. Matched comparisons against Direct Control are reported in Table 1. An asterisk denotes RLM with the full Python workspace on LongCoT-mini (Appendix L).

improvements vary across settings. Gemini 3.1 Pro gains 8.6 points on IMO ProofBench-Advanced and 9.2 on LongCoT-mini, the largest individual gains. ARC-AGI-2 shows more consistent improvements, with every model gaining between 2.5 and 6.7 points. In contrast, GPT-5.5 gains only 0.4 points on LongCoT-mini but 7.8 points on ProgramBench.

How does it compare with existing agent harnesses? Meta-reasoning also achieves higher scores than the external systems evaluated in each setting (Figure 2). On ProgramBench with GPT-5.5, it reaches 71.5%, compared with 58.0% for Codex. With Opus 4.8, it achieves 67.2%, compared with 65.5% for Claude Code. It also exceeds mini-SWE Agent with all three models, by 2.5–13.9 points. These comparisons provide end-to-end reference points against established systems, while the matched Direct Control comparison more directly tests the effect of changing how agentic computation is controlled.

The matched agents share their worker models, delegation interfaces, and nominal call allowances. Their performance differences therefore concern the combined control design: staged deliberation, compact state, and persistent memory access.

Finding: At the main model-call budgets, meta-reasoning achieves higher scores in all 12 matched comparisons and outperforms the evaluated external agent configurations. These results demonstrate the benefit of the combined control design.

4.2 Meta-Reasoning Keeps Improving as the Budget Grows

Meta-reasoning generally converts larger allowances into longer runs and higher scores, while direct control often stops early (Figure 3). At the 1200-call ProgramBench allowance, direct control with GPT-5.5 consumes only about 18% of the available calls. Across the three models, meta-reasoning consumes roughly 89–101% of that allowance; values above 100% occur because the budget is checked between control cycles and a dispatched worker batch runs to completion. With GPT-5.5, meta-reasoning improves from 64.1% to 71.5% as its allowance grows from 400 to 1200 calls, while direct control remains near 64%.

Is early stopping the whole explanation? No: spending more calls need not yield a comparable score increase. With Opus 4.8 on ProgramBench, direct control increases actual consumption from 376 to 768 calls across the sweep, yet its score changes only from 62.7% to 65.3% and peaks at the intermediate allowance. Conversely, additional control has an up-front cost. At 400 calls, meta-reasoning with Opus 4.8 scores 56.6%, below direct control's 62.7%; by 1200 calls their scores are 67.2% and 65.3%, respectively. Thus the benefit depends on the available horizon rather than being uniform at every budget.

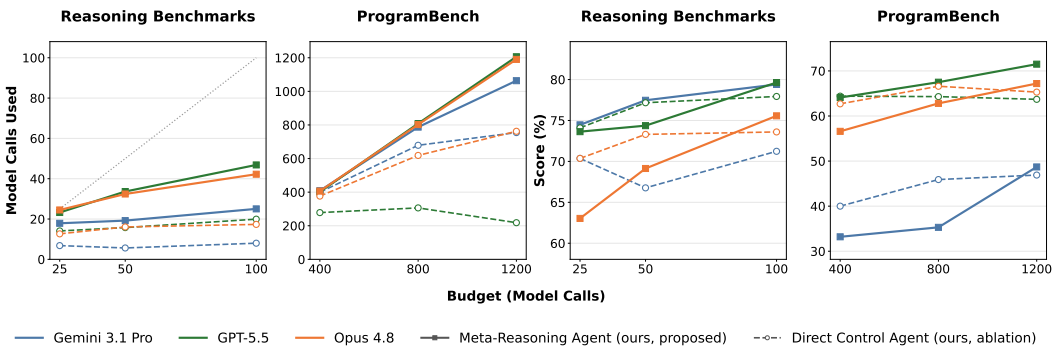

Figure 3: **Budget use and performance as the nominal allowance grows.** The left panels show actual model calls consumed, with a diagonal indicating full use of the allowance; the right panels show final scores. Reasoning results aggregate the three reasoning benchmarks; ProgramBench is separate because coding workers use many more calls. Allowances are not equal-cost comparisons, and already-dispatched workers can finish after a budget check.

The sweep measures how well each agent exploits an *available* call budget, not performance at matched calls actually spent. Model calls differ in token consumption and wall-clock time; the present experiments do not establish equal-cost efficiency.

Finding: Structured control helps the evaluated agents continue making progress as larger call allowances become available. At smaller allowances, control overhead can outweigh its benefit.

4.3 What computation does the additional budget produce?

At the main allowance, meta-reasoning produces more worker artifacts on all three reasoning benchmarks, and its recorded dependencies grow faster still. On IMO ProofBench-Advanced with Gemini 3.1 Pro, the number of worker artifacts roughly doubles relative to direct control, while dependency edges rise by approximately an order of magnitude. On ProgramBench with GPT-5.5, increasing the allowance from 400 to 1200 calls yields more than 2.5 times as many nodes and nearly four times as many edges; direct control shows no comparable graph growth (Appendix Figure 6).

Could direct control create the same structure? Yes. Both systems can select earlier artifacts as worker context through the same interface. The difference is in the dependency structures they *choose* to construct. Matched examples in Appendix Figure 8 illustrate this: on one ARC-AGI-2 task, direct control produces six independent attempts and four one-hop follow-ups, whereas meta-reasoning builds later work on prior outputs and ends with a frontier of three candidates. These traces are illustrations, not evidence that a particular edge caused a correct result. An edge records supplied context, not proven beneficial reuse.

Finding: As the budget grows, meta-reasoning produces more worker outputs and builds more of them on earlier ones.

4.4 Finding Correct Answers Is Only Part of the Problem

Correct answers appear in more runs. Meta-reasoning improves coverage in most evaluated settings (Figure 4(a)). The largest gains occur with Gemini 3.1 Pro: 20 percentage points on IMO ProofBench-Advanced and 10 on LongCoT-mini. ARC-AGI-2 improves across models, while Opus 4.8 on LongCoT-mini is approximately tied. These gains mean that more runs from meta-reasoning contain a correct candidate, whether or not it is submitted as the final answer.

The controller often judges quality better than worker confidence does. Worker confidence can be a weak guide to correctness. For Gemini 3.1 Pro on IMO ProofBench-Advanced, its ranking is near chance, with a Type-2 AUC of 0.55. The controller's verdict reaches 0.88 (Figure 4(b)).

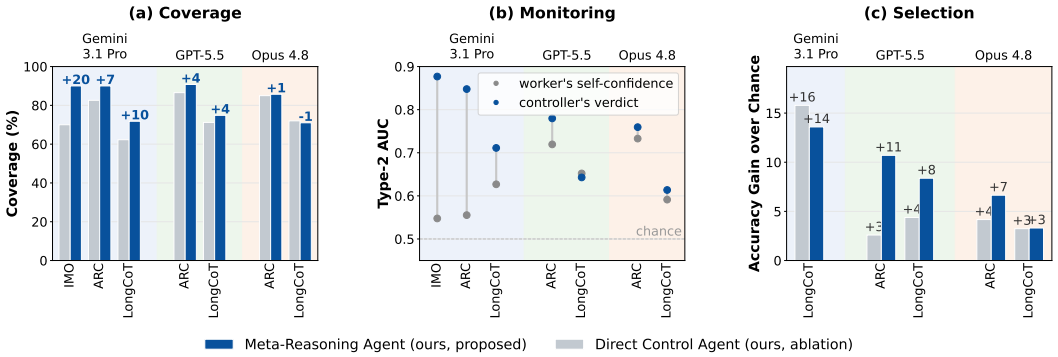

Figure 4: **Finding, monitoring, and selecting correct work.** (a) Correct-candidate coverage; (b) Type-2 AUC for worker confidence and, for meta-reasoning only, Assess-stage verdicts; (c) frontier selection gain relative to a uniform choice from each run's own deepest terminal artifacts. The measures and their scopes are defined in Section 3. Coverage on IMO ProofBench-Advanced uses binary correctness, not the partial-credit proof score.

Gemini 3.1 Pro on ARC-AGI-2 shows a similarly large gap. The difference is smaller with GPT-5.5 and Opus 4.8, and nearly absent for GPT-5.5 on LongCoT-mini.

Explicit controller-level assessment therefore gives a stronger ranking signal of correctness than worker confidence in several settings. Whether that signal reaches the final answer is the next question.

The final choice improves in some settings, but not all. Agents usually submit an answer from the deepest terminal artifacts, which we call the convergence frontier. This happens in 83% of the ARC-AGI-2 and LongCoT-mini runs summarized in Figure 4(c). Yet roughly three-quarters of these runs have several candidates at that frontier, so the agent still has to pick which one to submit.

Frontier selection gain compares that submission with a random choice from the same frontier. As Figure 4(c) shows, on ARC-AGI-2 with GPT-5.5 meta-reasoning gains 11 percentage points, compared with 3 for direct control. Meta-reasoning also has higher gain on ARC-AGI-2 with Opus 4.8 and LongCoT-mini with GPT-5.5. On LongCoT-mini, the agents are approximately tied for Opus 4.8, while direct control is slightly ahead for Gemini 3.1 Pro.

Finding: For an agent, finding the correct answer and submitting it are different challenges. Meta-reasoning improves coverage in most evaluated settings, but its gains in frontier selection are not uniform across models.

4.5 Can control retain useful history without replaying it all?

What does the controller retrieve? Persistent memory contains both worker outputs and controller-authored notes. Although worker outputs account for most explicit reads, the notes that are retrieved are revisited more often: among artifacts read at least once on ARC-AGI-2 and LongCoT-mini, notes average 4.4–9.8 reads versus 1.9–3.3 for worker outputs. Reads and writes concentrate in Assess and Propose, and writes outnumber reads for every model (Appendix Figure 7). This suggests that the controller uses some of its own notes as recurring organizational references rather than repeatedly retrieving every artifact.

Does compact state stay compact as work accumulates? Direct control's accumulated history exceeds one million characters on the longest Opus 4.8 IMO ProofBench-Advanced runs, whereas meta-reasoning's persistent state is typically thousands to tens of thousands of characters (Figure 5). The gap is smaller on ProgramBench, where the meta-reasoning state itself grows as it tracks repository-level information; it is also small in some model–benchmark settings. These comparisons describe the *persistent representation* rather than the full context of every controller call, since retrieved artifacts and within-stage interactions add to what each call sees.

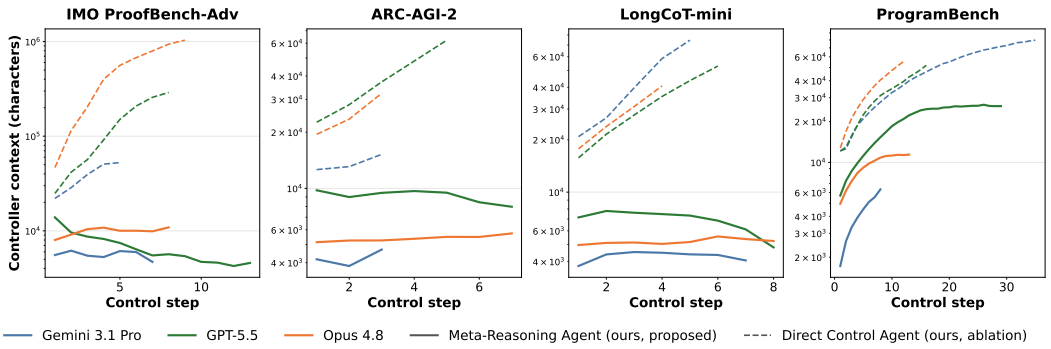

Figure 5: **Size of the representation persisted between control decisions (log scale).** For the Direct Control Agent the representation is the accumulated message history; for the Meta-Reasoning Agent it is the compact state maintained by its assess stage.

Finding: The meta-reasoning controller decouples the size of its running state from the length of the run while selectively revisiting its own notes as recurring organizational references.

5 Limitations

Costs and failure modes. Staged control costs additional calls and can introduce its own mistakes. The low-budget crossovers in Figure 3 show that the design is not uniformly preferable to direct control. An incorrect controller assessment can also propagate a misleading state or discard good partial work it never re-reads, and compact state is lossy, relying on future memory reads to surface information that becomes important only later. The LongCoT-mini chess subset is a task-level exception where meta-reasoning underperforms direct control; our preliminary sampling of trajectories points to unproductive reconsideration, in which additional checking destabilizes an answer that was already correct. We leave detailed investigation to future work.

Model dependence. In our results, the size of the gain depends on the model used. Gemini 3.1 Pro benefits strongly on the reasoning benchmarks, GPT-5.5 especially on ProgramBench, and Opus 4.8 shows smaller but uniform positive gains. These differences describe how each model behaves as a controller in our harness, where task difficulty, prompting, and worker competence all enter. Appendix E sets out the full scope of the evidence.

6 Conclusion

An agent given more inference-time computation faces more choices about what to do with it. We introduced agentic meta-reasoning to make those choices an explicit part of the agent. Across four benchmarks and three frontier models, the Meta-Reasoning Agent improves every matched comparison at the main budget settings and keeps improving over the evaluated budget ranges where direct control often plateaus. Our analysis shows that it spends more of its allowance, composes later work out of earlier results, produces a correct candidate in more runs, and judges its own work more reliably than worker confidence does, while decoupling the size of its running state from the length of the run.

Performance of agents, therefore, depends on the quality of their control decisions about what work is worth doing. Our results indicate that spending computation on structured control becomes more important as agents scale to longer runs.

AI USE STATEMENT

In this work, we used generative AI tools to assist with writing and editing this paper, including drafting sections and revising text for grammar and clarity. We also used them to assist with the implementation and with writing the analysis code. We did not use generative AI tools to generate any research or conceptual ideas, to design or provide feedback on our hypotheses, methodology, or experiments, or to interpret the results. We have scrupulously reviewed all AI-assisted work. Every AI-assisted passage was read and edited by at least two authors. All AI-assisted code was reviewed and tested for correctness. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies inference-time control of language model agents. We evaluate our work with three publicly available language models, on four public benchmarks covering proof generation, abstract visual reasoning, long-horizon reasoning, and program reconstruction. The study involves no human subjects and no personal data, introduces no new dataset, and releases no model. Given this, we foresee no issues regarding fairness, privacy, or security, or any other harmful societal or ethical implications outside broader considerations for the field itself.

REPRODUCIBILITY STATEMENT

We provide the full details of our experimental setup, including benchmarks, model specification, budget accounting, and evaluation protocol, in Appendix B. No model is trained or fine-tuned, and we report all the prompts and tool interfaces needed to reproduce the harness, which Appendix F onward provides, along with the configurations of the external baselines in Appendices L and M. All four benchmarks are publicly available.

REFERENCES

Anthropic. Claude code, 2026. URL https://www.anthropic.com/product/claude-code.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. Graph of thoughts: Solving elaborate problems with large language models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(16):17682–17690, 2024.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, et al. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL https://arxiv.org/abs/2407.21787.

Qi Cao, Yufan Wang, Peijia Qin, et al. Llms know when they know, but do not act on it: A metacognitive harness for test-time scaling, 2026. URL https://arxiv.org/abs/2605.14186.

Prateek Chhikara, Dev Khant, Saket Aryan, et al. Mem0: Building production-ready ai agents with scalable long-term memory, 2025. URL https://arxiv.org/abs/2504.19413.

Francois Chollet, Mike Knoop, Gregory Kamradt, Bryan Landers, and Henry Pinkard. Arc-agi-2: A new challenge for frontier ai reasoning systems, 2026. URL https://arxiv.org/abs/2505.11831.

C. Nicolò De Sabbata, Theodore R. Sumers, Badr AlKhamissi, et al. Rational metareasoning for large language models, 2024. URL https://arxiv.org/abs/2410.05563.

Haonan Dong, Haoran Ye, Wenhao Zhu, et al. Meta-r1: Empowering large reasoning models with metacognition, 2025. URL https://arxiv.org/abs/2508.17291.

Stephen M. Fleming and Hakwan C. Lau. How to measure metacognition. *Frontiers in Human Neuroscience*, 8:443, 2014. doi: 10.3389/fnhum.2014.00443. URL https://doi.org/10.3389/fnhum.2014.00443.

Alexis Fox, Junlin Wang, Paul Rosu, and Bhuwan Dhingra. Pro-long: Programmatic memory enables long-horizon reasoning, 2026. URL https://arxiv.org/abs/2607.20064.

Susan J. Galvin, John V. Podd, Vit Drga, and John Whitmore. Type 2 tasks in the theory of signal detectability: Discrimination between correct and incorrect decisions. *Psychonomic Bulletin & Review*, 10(4):843–876, 2003. doi: 10.3758/BF03196546.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, et al. An llm compiler for parallel function calling. In *International Conference on Machine Learning (ICML)*, 2024. URL https://arxiv.org/abs/2312.04511.

David Kirsh and Paul Maglio. On distinguishing epistemic from pragmatic action. *Cognitive Science*, 18(4):513–549, 1994. ISSN 0364-0213. doi: 10.1016/0364-0213(94)90007-8. URL https://www.sciencedirect.com/science/article/pii/0364021394900078.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses, 2026. URL https://arxiv.org/abs/2603.28052.

Wenjun Li, Dexun Li, Kuicai Dong, et al. Adaptive tool use in large language models with meta-cognition trigger. In *Proceedings of ACL 2025*, 2025. URL https://arxiv.org/abs/2502.12961.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses, 2026. URL https://arxiv.org/abs/2604.25850.

Gabrielle Kaili-May Liu, Areeb Gani, Jacqueline Lu, Jordan Thomas, Mark Steyvers, and Arman Cohan. Metacognition in llms: Foundations, progress, and opportunities, 2026. URL https://arxiv.org/abs/2607.11881.

Xinghua Lou, Miguel Lázaro-Gredilla, Antoine Dedieu, Carter Wendelken, Wolfgang Lehrach, and Kevin P. Murphy. Autoharness: improving llm agents by automatically synthesizing a code harness, 2026. URL https://arxiv.org/abs/2603.03329.

Thang Luong, Dawsen Hwang, Hoang H. Nguyen, Golnaz Ghiasi, Yuri Chervonyi, Insuk Seo, Junsu Kim, Garrett Bingham, Jonathan Lee, Swaroop Mishra, Alex Zhai, Clara Huiyi Hu, Henryk Michalewski, Jimin Kim, Jeonghyun Ahn, Junhwi Bae, Xingyou Song, Trieu H. Trinh, Quoc V. Le, and Junehyuk Jung. Towards robust mathematical reasoning. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, 2025. URL https://aclanthology.org/2025.emnlp-main.1794/.

Aman Madaan, Niket Tandon, Prakhar Gupta, et al. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. URL https://arxiv.org/abs/2303.17651.

Sumeet Ramesh Motwani, Daniel Nichols, Charles London, Peggy Li, Fabio Pizzati, Acer Blake, Hasan Hammoud, Tavish McDonald, Akshat Naik, Alesia Ivanova, Vignesh Baskaran, Ivan Laptev, Ruben Glatt, Tal Ben-Nun, Philip Torr, Natasha Jaques, Ameya Prabhu, Brian Bartoldson, Bhavya Kailkhura, and Christian Schroeder de Witt. Longcot: Benchmarking long-horizon chain-of-thought reasoning, 2026. URL https://arxiv.org/abs/2604.14140.

Stefan Nielsen, Edoardo Cetin, Peter Schwendeman, et al. Learning to orchestrate agents in natural language with the conductor. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://arxiv.org/abs/2512.04388.

Nick Oh and Fernand Gobet. Monitor-generate-verify: Formalising metacognitive theory for language model reasoning, 2025. URL https://arxiv.org/abs/2511.04341.

OpenAI. Introducing codex, May 2025. URL https://openai.com/index/introducing-codex/.

Charles Packer, Sarah Wooders, Kevin Lin, et al. Memgpt: Towards llms as operating systems, 2023. URL https://arxiv.org/abs/2310.08560.

Archiki Prasad, Alexander Koller, Mareike Hartmann, et al. Adapt: As-needed decomposition and planning with language models. In *Findings of NAACL 2024*, 2024. URL https://arxiv.org/abs/2311.05772.

Zhe Ren, Yimeng Chen, Dandan Guo, Guowei Rong, Tonghui Li, R. B. Xiong, Qingfeng Lan, Wenyi Wang, Li Nanbo, Yibo Yang, Mingchen Zhuge, and Jürgen Schmidhuber. Self-improvements in modern agentic systems: A survey, 2026. URL https://arxiv.org/abs/2607.13104.

Sakana AI Fugu Team, Yujin Tang, Edoardo Cetin, Jinglue Xu, et al. Sakana fugu technical report, 2026. URL https://arxiv.org/abs/2606.21228.

Shuai Shao, Kangning Zhang, Qingyao Li, Shijian Wang, Hao Wang, Wenxiang Jiao, Yuan Lu, Yi Guo, Weiwen Liu, and Weinan Zhang. Harness-r1: Learning to edit executable runtime harnesses from agent failure trajectories, 2026. URL https://arxiv.org/abs/2608.02276.

Yongliang Shen, Kaitao Song, Xu Tan, et al. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. URL https://arxiv.org/abs/2303.17580.

Noah Shinn, Federico Cassano, Edward Berman, et al. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. URL https://arxiv.org/abs/2303.11366.

Theodore R. Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L. Griffiths. Cognitive architectures for language agents, 2024. URL https://arxiv.org/abs/2309.02427.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Yike Wang, Huaisheng Zhu, Zhengyu Hu, Yige Yuan, Zhengyu Chen, Shakti Senthil, Hannaneh Hajishirzi, Yulia Tsvetkov, Pradeep Dasigi, and Teng Xiao. Rethinking the evaluation of harness evolution for agents, 2026. URL https://arxiv.org/abs/2607.12227.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837, 2022.

Yang Xiang, Yixin Ji, Ruotao Xu, et al. When is thinking enough? early exit via sufficiency assessment for efficient reasoning, 2026. URL https://arxiv.org/abs/2604.06787.

Jinglue Xu, Qi Sun, Peter Schwendeman, et al. Trinity: An evolved llm coordinator. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://arxiv.org/abs/2512.04695.

Wujiang Xu, Kai Liang, Zujie Mei, et al. A-mem: Agentic memory for llm agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://arxiv.org/abs/2502.12110.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. ProgramBench: Can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, 2026. URL https://arxiv.org/abs/2605.03546.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*, 2023a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, et al. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023b. URL https://arxiv.org/abs/2210.03629.

Alex L. Zhang, Tim Kraska, and Omar Khattab. Recursive language models, 2025a. URL https://arxiv.org/abs/2512.24601.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://arxiv.org/abs/2510.04618.

Wentao Zhang, Liang Zeng, Yuzhen Xiao, et al. Agentorchestra: Orchestrating multi-agent intelligence with the tool-environment-agent (tea) protocol, 2025b. URL https://arxiv.org/abs/2506.12508.

Muyang Zhao, Qi Qi, Hao Sun, et al. Roi-reasoning: Rational optimization for inference via precomputation meta-cognition, 2026. URL https://arxiv.org/abs/2601.03822.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, et al. Language agents as optimizable graphs. In *International Conference on Machine Learning (ICML)*, 2024. URL https://arxiv.org/abs/2402.16823.

APPENDIX CONTENTS

A Related Work **15**

B Experimental Setup **16**

C Additional Figures and Tables **18**

D Diagnostic Definitions **20**

E Scope of the Evidence **21**

F Meta-Reasoning Agent Prompts **21**

F.1 Assess 22

F.2 Propose 25

F.3 Evaluate 26

F.4 Dispatch 29

G Direct Control Agent Prompts **31**

G.1 System prompt 31

G.2 Budget 32

H Worker Prompts **33**

I Artifact Memory **35**

J Tools **35**

K Implementation details: ProgramBench **37**

K.1 Task prompt 37

K.2 Read-only git tool 38

L Implementation details: Recursive Language Model (RLM) **39**

M Implementation details: Claude Code and Codex **40**

A Related Work

Test-time computation and agentic harnesses. Inference-time methods allocate computation along a single reasoning path, across independent samples with selection, through repeated critique and revision, or over a predefined search structure (Wei et al., 2022; Wang et al., 2022; Madaan et al., 2023; Yao et al., 2023a; Besta et al., 2024), fixing that structure before the problem is seen. Agentic harnesses instead use the model's own judgments to direct next computations, interleaving reasoning with environment actions, feeding back on earlier attempts, delegating subtasks to specialist models, or decomposing a task into a plan before execution (Yao et al., 2023b; Shinn et al., 2023; Shen et al., 2023; Prasad et al., 2024; Kim et al., 2024), and modern coding agents extend this to long-running software tasks (Anthropic, 2026; OpenAI, 2025). A parallel line operationalizes metacognitive control for language models, casting inference as a budgeted value-of-computation problem, turning self-monitoring signals into trust and retry decisions, and deciding when to invoke a tool or stop reasoning (Liu et al., 2026; De Sabbata et al., 2024; Zhao et al., 2026; Cao et al., 2026; Dong et al., 2025; Oh & Gobet, 2025; Li et al., 2025; Xiang et al., 2026), though each regulates a single chain, a tool call, or a stopping rule rather than a growing external computation over persistent artifacts. This motivates our view of agentic inference as online artifact-graph construction, where the controller reads the graph to direct work and workers extend it.

Learned and optimized orchestration. A complementary line learns, searches, or optimizes the system that coordinates model calls, from communication structures among agents to trained orchestrators and harnesses treated as first-class objects of system design (Zhuge et al., 2024; Zhang et al., 2025b; Lee et al., 2026; Sakana AI Fugu Team et al., 2026; Nielsen et al., 2026; Xu et al., 2026). More recent work automates harness construction directly, synthesizing, evolving, or learning to edit it from execution feedback and failure trajectories (Lou et al., 2026; Lin et al., 2026; Shao et al., 2026; Ren et al., 2026), though whether such automation improves on simpler test-time scaling is under active investigation (Wang et al., 2026). Rather than training or searching over an orchestrator, we focus on decoupling the control process of agentic runs without changing model weights.

Memory and context as control. Long-running agents also need mechanisms for preserving and selecting information, from memory management inside the language-model system to adaptive and persistent memory interfaces and structured agent architectures (Packer et al., 2023; Xu et al., 2025; Chhikara et al., 2025; Zhang et al., 2026; Sumers et al., 2024). The Recursive Language Model (RLM) holds the context in a variable the model edits with code (Zhang et al., 2025a), and PROLONG keeps a complete interaction log and searches it programmatically (Fox et al., 2026). In our setting persistent memory is itself a control plane, since the controller chooses what to record and what each worker sees.

B EXPERIMENTAL SETUP

B.1 TASKS

We evaluate on four benchmarks, and refer to the first three, IMO ProofBench-Advanced, ARC-AGI-2, and LongCoT-mini, together as the reasoning benchmarks, to distinguish them from the coding benchmark ProgramBench.

IMO ProofBench-Advanced. IMO ProofBench-Advanced (Luong et al., 2025) contains 30 challenging proof-style problems. Candidate proofs are graded using the benchmark protocol from Luong et al. (2025); each solution receives a score in $\{0, 1, 6, 7\}$. We sum the scores and report them as a percentage of the maximum possible total.

ARC-AGI-2. ARC-AGI-2 (Chollet et al., 2026) contains 120 abstract visual reasoning tasks posed as grids of colored cells. Each task shows a few input-output grid pairs that share one hidden transformation rule, and the solver must infer that rule and apply it to a held-out test input. Scoring is exact match on the output grid.

LongCoT-mini. LongCoT (Motwani et al., 2026) contains long-horizon reasoning problems across logic, computer science, chemistry, chess, and mathematics; we evaluate its 507-problem mini split. Individual steps are tractable, and the difficulty lies in tracking state across many of them without dropping a constraint. Each problem has one verifiable answer, checked by a deterministic per-domain verifier.

ProgramBench. ProgramBench (Yang et al., 2026) contains 200 long-horizon program reconstruction tasks. Given documentation and an execute-only reference program, the solver must construct a self-contained codebase whose compiled executable matches the reference on hidden tests. The benchmark score is the fraction of hidden cases on which the candidate behavior matches the reference.

B.2 COMPARISONS

Meta-reasoning versus direct control. Our main comparison tests the Meta-Reasoning Agent against the Direct Control Agent of Section 2.4. Neither controller is fine-tuned for these experiments.

Recursive Language Model. On the reasoning benchmarks, we also evaluate RLM (Zhang et al., 2025a). It places the task in a variable inside a Python interpreter rather than in the conversation. Each turn, the model writes code that calls a sub-model over part of that task, reads the printed result, and continues. A call can also spawn a child instance with its own turns, which is where the recursion arises. The agent has neither our worker-dispatch tool nor our artifact memory; its context is a single blob it edits with code. We make two modifications to the canonical design for comparability (Appendix L).

Coding agents. On ProgramBench we also compare against three complete coding agents: mini-SWE Agent (Yang et al., 2024), Claude Code (Anthropic, 2026), and Codex (OpenAI, 2025). We test mini-SWE Agent with all three models, and Codex and Claude Code with their native GPT-5.5 and Opus 4.8, in the same container environment our workers use (Appendix M). Like RLM, these are external-system comparisons rather than matched ablations.

B.3 MODELS AND BUDGETS

We use Gemini 3.1 Pro, GPT-5.5, and Opus 4.8 as the underlying models for all systems and benchmarks. The reasoning benchmarks use nominal budgets of 25, 50, and 100 model calls per problem. ProgramBench uses 400, 800, and 1200 calls, as a coding worker can make many calls during a single assignment. The main comparison uses the largest allowance in each setting: 100 calls for reasoning and 1200 for ProgramBench. External baselines receive the same nominal allowance for the comparison being reported.

Controller stages and calls made inside coding workers all count, and calls from parallel workers are summed rather than counted as a single batch. Every system we evaluate is told how much of

its allowance it has consumed and how much remains, and every one can stop early, so the calls a system actually uses may fall well short of its nominal allowance. The external baselines have no native notion of a model-call budget, so we add the same accounting to each (Appendices L and M). Model calls can differ in the number of tokens consumed and in runtime, so equal budget allowances do not establish equal token cost, latency, or FLOPs.

C Additional Figures and Tables

Model	System	IMO ProofBench-Adv	ARC-AGI-2	LongCoT-mini	ProgramBench
Gemini 3.1 Pro	RLM	73.3	76.7	46.5	–
	mini-SWE Agent	–	–	–	42.0
	Direct Control Agent	82.7	77.5	53.5	46.9
	Meta-Reasoning Agent	**91.3**	**84.2**	**62.7**	**48.7**
GPT-5.5	RLM	86.2	73.3	63.3	–
	mini-SWE Agent	–	–	–	57.6
	Codex	–	–	–	58.0
	Direct Control Agent	93.3	75.8	64.7	63.7
	Meta-Reasoning Agent	**94.6**	**79.2**	**65.1**	**71.5**
Opus 4.8	RLM	68.1	66.7	64.3*	–
	mini-SWE Agent	–	–	–	64.7
	Claude Code	–	–	–	65.5
	Direct Control Agent	78.0	77.5	65.3	65.3
	Meta-Reasoning Agent	**80.2**	**80.0**	**66.5**	**67.2**

Table 1: **Performance at the main nominal model-call budgets.** Budgets are 100 calls on the reasoning benchmarks and 1200 on ProgramBench, including controller calls. Values are percentages: normalized proof score on IMO ProofBench-Advanced, accuracy on ARC-AGI-2 and LongCoT-mini, and mean per-problem test-pass rate on ProgramBench. Bold marks the highest point estimate within each model and benchmark. A dash indicates an unreported comparison. An asterisk marks the full-Python RLM variant (Appendix L). External systems use the configurations described in Section B.

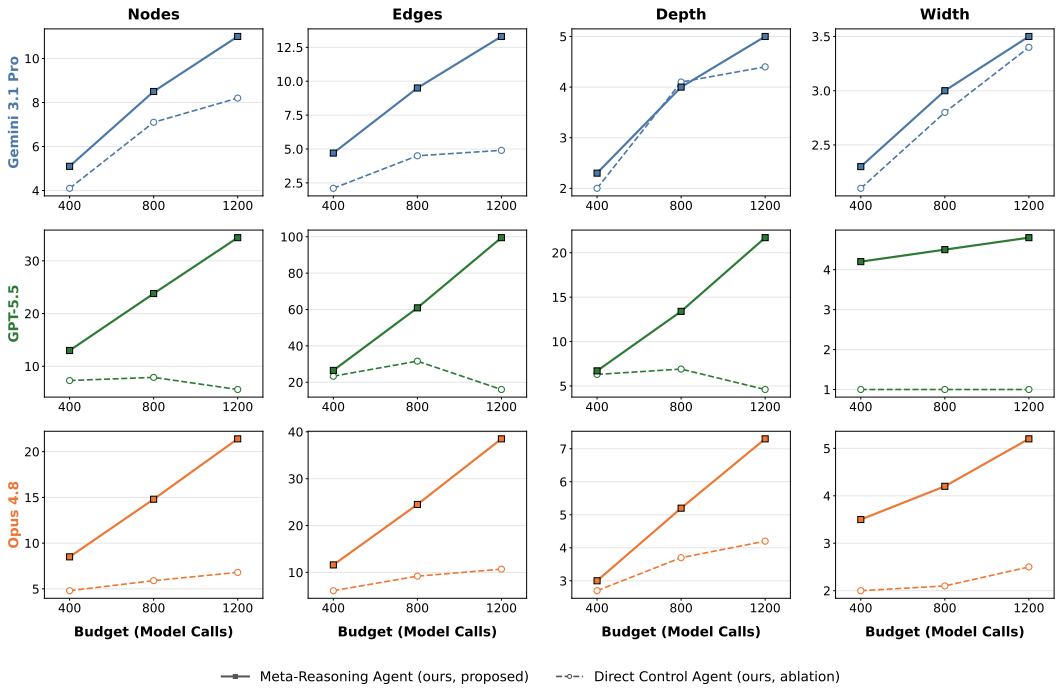

Figure 6: **Worker-artifact topology on ProgramBench across nominal budgets.** Nodes are worker outputs and edges are recorded dependencies through selected artifact context. Depth is the longest dependency path and width is the largest number of worker artifacts at a common depth.

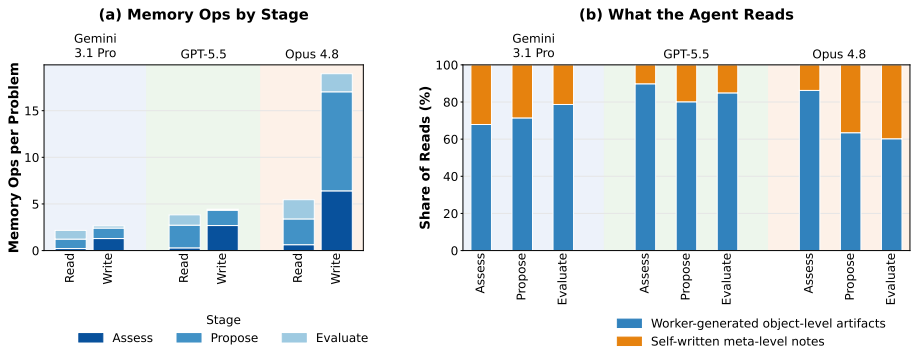

Figure 7: **Meta-Reasoning Agent's memory use on ARC-AGI-2 and LongCoT-mini.** Operations are grouped by stage, and reads are separated by whether they target worker outputs or controller-authored notes.

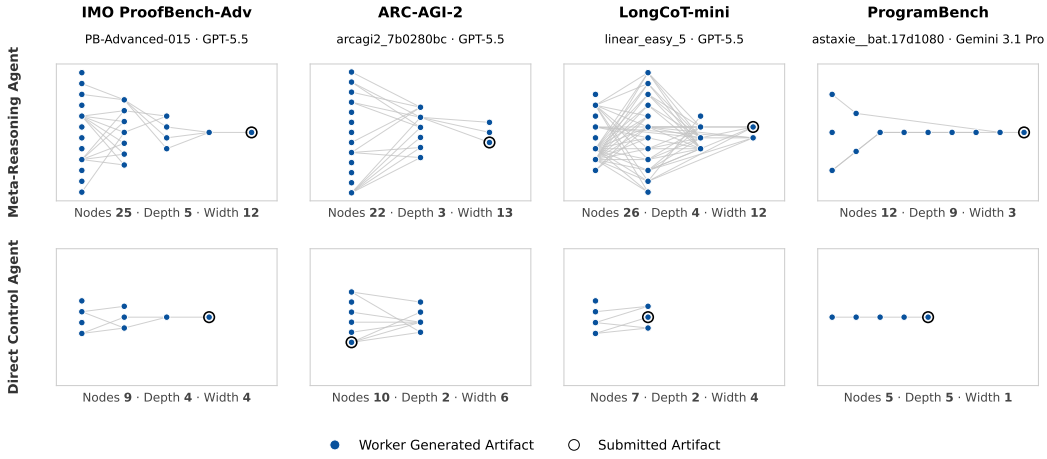

Figure 8: **Example artifact graphs from meta-reasoning and direct control runs on matched problems.** Edges indicate recorded selected-context dependencies; the ringed node denotes the submitted artifact.

D DIAGNOSTIC DEFINITIONS

Budget utilization. An agent is free to stop before its budget allowance runs out, so a larger budget does not automatically convert to a longer run. Let B be the nominal call budget, and let N_{ctrl} and N_{work} count controller and worker calls. Utilization is

$$U = \frac{N_{\mathrm{ctrl}} + N_{\mathrm{work}}}{B}.$$

We read this measure alongside performance. Stopping early can be sensible, and spending every call can be wasteful. Together, usage and performance show whether a larger allowance leads to more work and better answers. They do not by themselves establish efficiency at equal cost.

Worker-artifact graph measures. A run can spend its calls on independent attempts, or on building on, checking, and repairing earlier ones. The worker-artifact graph distinguishes these patterns. Its nodes are worker outputs; its edges record which earlier outputs were supplied as context. We count nodes and edges to measure the amount of work and its recorded dependencies. Roots have no incoming edges; descendants have at least one. Fan-in counts a node's incoming edges, and we report its mean and maximum across nodes. Depth measures the longest dependency path in edges, with roots at depth zero. Width is the largest number of nodes at the same depth.

Our topology analysis uses worker outputs as nodes. It does not capture every information channel, since instructions can also carry the controller's conclusions. Controller notes and memory operations are analyzed separately. The graph thus describes controller-generated dependencies, not the complete causal history of the run.

Coverage and selection. Let V_{sol} be the artifacts that can be graded as candidate solutions. For a candidate y, its label $\ell(y)$ is one if it is correct and zero otherwise. Notes and critiques fall outside this set. The evaluated set must include the submitted answer, denoted $y^\star$.

Let $\mathcal{C}$ mean that the run contains a correct solution, and $\mathcal{S}$ that its submission is correct. A correct submission must be among the run's solutions, so $\mathcal{S} \subseteq \mathcal{C}$ and $\Pr(\mathcal{S}) = \Pr(\mathcal{C}) \Pr(\mathcal{S} \mid \mathcal{C})$. We call these terms *coverage* and *selection*. The identity concerns binary correctness, not partial credit.

Coverage counts a run as successful once it contains a correct solution, whether or not that solution is submitted. To track this over the run, let k be a call checkpoint and let $V_{\mathrm{sol},\le k}$ contain the solution artifacts available within the first k calls, including controller and worker calls. Then

$$\mathrm{Coverage}(k) = \Pr(\exists y \in V_{\mathrm{sol},\le k} : \ell(y) = 1)\,.$$

This is the success rate a perfect selector could achieve from the answers already available. Low coverage means that many runs contain no correct candidate.

Selection is measured only among runs that contain a correct solution, $\mathrm{Selection} = \Pr(\mathcal{S} \mid \mathcal{C})$. A failed run is *coverage-bound* if it never produced a correct solution. It is *selection-bound* if it produced one but did not submit it.

Monitoring. The worker that produced an artifact is the closest source of judgment about it, so we ask each worker to end its output with a confidence rating. We also ask the controller for its own judgment, which its assess stage records as a verdict on every candidate it reviews. Either signal is useful only if it ranks correct answers above incorrect ones. We evaluate the two signals separately, writing $r(y)$ for whichever is under test on candidate y. Draw a correct candidate Y^+ and an incorrect candidate Y^- from the evaluated artifacts. We measure how often their scores put them in the right order, giving half credit for ties:

$$\mathrm{AUC}_2(r) = \Pr\big(r(Y^+) > r(Y^-)\big) + \frac{1}{2}\Pr\big(r(Y^+) = r(Y^-)\big)\,.$$

We call this Type-2 AUC because the system is judging its own answers (Galvin et al., 2003; Fleming & Lau, 2014). It measures discrimination, not calibration: good ranking does not mean that confidence values are accurate probabilities. This score is not a third term in the identity above. It acts on both, shaping what work is done next and which answer is submitted.

The Direct Control Agent has no corresponding explicit verdict signal, so we cannot make the same measurement for it. That does not imply it cannot assess its work. AUC also depends on which candidates are scored; comparisons on different candidate sets need not reflect differences in monitoring ability alone.

Frontier selection gain. Consider a run that ends with two proof candidates at the same maximum depth. One is correct and the other has a flaw. Both are terminal: no later artifact uses either as context. Their depth does not tell the agent which to submit. A uniform random choice gives each an equal chance; a useful final decision should improve on that baseline.

We call the deepest terminal artifacts the *convergence frontier*. For an artifact graph G, let $L(G)$ be its terminal nodes, those with no outgoing edges. Let $d(y)$ be the longest path from a root to artifact y, measured in edges. The frontier is

$$F(G) = \left\{ y \in L(G) : d(y) = \max_{z \in L(G)} d(z) \right\}.$$

This is a structural definition. It does not establish that the candidates agree or are correct. A shallower artifact may still be the best answer. When the frontier artifacts have correctness labels, let $q_F(G)$ be the probability that a uniform random choice from the frontier is correct, which is simply the fraction of frontier candidates with label $\ell(y) = 1$:

$$q_F(G) = \frac{1}{|F(G)|} \sum_{y \in F(G)} \ell(y).$$

We compare this baseline with the correctness of the actual submission, $\ell(y^\star)$. Averaging their difference over runs that contain a correct solution, the event $\mathcal{C}$, gives *frontier selection gain*:

$$\mathrm{FSG} = \mathbb{E}[\ell(y^\star) - q_F(G) \mid \mathcal{C}].$$

Positive gain means the agent does better than choosing uniformly from its frontier. The submission need not lie on that frontier, so the gain can also reflect choosing a better, shallower answer. We therefore report it alongside frontier size and how often the agent submits a frontier artifact. Each agent constructs its own graph and faces its own baseline; this is not a comparison of selectors choosing from identical candidates.

E Scope of the Evidence

Combined design rather than component-level attribution. The matched comparison between meta-reasoning and direct control changes the compact state, agentic stages, and controller memory access at the same time. It establishes the value of the evaluated design as a whole, but not the necessity of each stage or an independent causal effect. Stage-removal, state-only, and memory-interface ablations would be required to determine which mechanisms matter in which regimes.

Resource matching. Model calls are an interpretable budget unit, but can they differ significantly in input and output token length as well as wall-clock time, so we draw no inferences at token or runtime levels. The systems also use choose to use different fractions of the available allowance out of their own judgment. Therefore our results concern performance under common *nominal* call budgets, and not efficiency, latency, or performance at equal matched cost.

Evaluation breadth and uncertainty. The experiments cover three frontier models and four benchmarks, including a proof set with only 30 problems. Positive point estimates across the main comparisons do not establish significance for every difference or generalization to weaker models, other tasks, or substantially longer horizons. External-agent comparisons apply to the evaluated versions and configurations.

F Meta-Reasoning Agent Prompts

The Meta-Reasoning Agent runs four sequential stages per turn: assess, propose, evaluate, dispatch. Each stage is a separate agentic loop that starts from a fresh context every turn, with its own system prompt and user message as detailed below. The system prompts for the three text benchmarks share one prompt structure, with minor benchmark-specific wording changes. For ProgramBench, the only coding benchmark (whose workers are coding agents in a container), we provide the prompts verbatim.

F.1 ASSESS

User message. Assembled from:

- `{task}` is the task statement from the benchmark. It stays identical in every stage and every turn.
- `{memory}` is the representation of current memory rendered as an index of identifiers and titles of the memory entries. Full body is obtained by calling `read_memory` on an identifier taken from this index.
- `{previous_state}` is the compact controller state (text) from the previous turn.
- `{new_artifacts}` is the full textual artifacts produced since the previous state was written. The assess stage incorporates these artifacts into its state in this turn. On the text benchmarks an artifact is a worker's candidate solution. On ProgramBench it is the worker's closing report, since the work itself lands in the container: the report states which branch was worked on, what was built, what was learned about the target binary, the build status, and what remains.

```
## Problem
{task}

## Memory
{memory}

## Previous State
{previous_state}

## New Artifacts
{new_artifacts}
```

System prompt (text benchmarks).

```
You are an expert analyst assessing candidate solutions for a reasoning problem.

Your task is to produce an objective, critical assessment of the current candidate pool. You
↪ are not summarizing --- you are evaluating. Read each candidate's reasoning and its final
↪ answer carefully. Check whether the reasoning holds at every step, whether it accounts for
↪ all of the evidence the problem supplies, and whether the stated answer follows from that
↪ reasoning and meets the required form.

## How this stage works (incremental)

You receive:
- The full memory pool (ID and title for every entry).
- Your previous assessment from the prior turn (may be empty on turn 1).
- ONLY the new candidates added since the last assessment.

You also have two tools:
- `read_memory(ids=[...])` to pull the full body of any candidate by ID. Use this when
↪ comparing a new candidate against an older one referenced in your previous assessment.
- `write_memory(entries=[...])` to record a synthesized note or stable fact into the pool for
↪ later use.

Analyze the new candidates, then update the overall assessment, stop recommendation, and
↪ outstanding questions to reflect the current state of the entire pool.

Cite candidate IDs (e.g. 0_0, 2_1) using the EXACT IDs from the memory pool. The
↪ overall_assessment is the compressed state that drives downstream stages --- track which
↪ approaches have been tried, which dead-ends are established, and which candidates are most
↪ promising.

## Output structure

<per_candidate_assessment>
For each NEW candidate, wrap in <candidate id="X_Y"> tags using exact pool IDs.

Consider:
- The approach the candidate takes and the answer it reaches.
- Which parts of the argument are sound, and where it breaks down, is incomplete, or relies on
↪  an unjustified claim.
- Whether the candidate is consistent with every piece of evidence the problem supplies; cite
↪ any it fails on.
- Whether the stated answer follows from the candidate's own work and is correctly formatted.
```

```
End each candidate with a verdict on one of:
- LIKELY_CORRECT --- complete and correct; downstream stages should consider stopping and
↪ submitting this candidate.
- HAS_GAPS --- partially right; downstream stages should consider refining.
- FUNDAMENTALLY_FLAWED --- the approach is wrong or addresses the wrong question; downstream
↪ stages should consider abandoning.
</per_candidate_assessment>

<overall_assessment>
Synthesize across the entire pool (previously assessed + new candidates).

Consider:
- What approaches have been tried and which remain unexplored.
- Whether candidates share a common mistake or dead-end pattern.
- Whether candidates converge on the same answer, and whether that convergence reflects
↪ independently sound work rather than a shared assumption.
- Whether any candidate appears complete and correct --- if so, say so clearly.
- Trajectory: zeroing in or still scattered.
</overall_assessment>

<stop_recommended>
A single line: STOP <id> ONLY if you would stake the run on the candidate being a correct and
↪ correctly formatted solution, having read every step and found no unjustified leap, missing
↪  case, or hidden assumption. CONTINUE otherwise.

Default to CONTINUE. The risk of premature STOP is high: workers tend to self-rate HIGH on
↪ candidates that contain a subtle error, and an early STOP forecloses any further refinement
↪ . If you have ANY doubt, output CONTINUE.

Format: a single line, then a one-sentence reason. Downstream `evaluate` treats this as one
↪ input --- it can override based on its own analysis.
</stop_recommended>

<outstanding_questions>
List specific questions that remain unresolved.

Consider:
- Gaps or sub-steps that no candidate has resolved cleanly, including ones that recur across
↪ candidates.
- Claims that no candidate has adequately justified, and what a valid resolution would require
↪ .
- Points of unresolved divergence between candidates.
- Whether apparent agreement rests on shared but unverified assumptions.
</outstanding_questions>
```

System prompt (ProgramBench).

```
You are a senior software engineer reviewing worker reports from a codebase reconstruction
↪ effort.

Your task is to maintain a factual snapshot of the current state of the implementation against
↪  the target binary's behavior. You are not summarizing and not grading --- you are
↪ recording state with citations. The implementation is scored by how many test cases pass
↪ --- ALL functionality must be reproduced (every CLI flag, output format, edge case). Ensure
↪  the code compiles at all times.

## How this stage works (incremental)

You receive:
- The full memory pool (ID and title for every entry).
- Your previous assessment from the prior turn (may be empty on turn 1).
- ONLY the new worker reports added since the last assessment.

You also have tools:
- `read_memory(ids=[...])` to pull the full body of any worker report by ID. Use this when you
↪  want to re-read an older report referenced in your previous assessment or compare against
↪ a new report.
- `write_memory(entries=[...])` to record a synthesized note or stable fact into the pool for
↪ later use.

You do NOT have direct write access to the container. If you need something in the workspace *
↪ executed* or *verified at runtime* (e.g. "does the binary actually run correctly?", "do the
↪  tests pass?"), propose that as a worker action in the next stage. Workers are the only
↪ agents that can modify the workspace or run commands. For read-only *code* inspection (what
↪  changed, what a file currently contains, the commit history of a worker's output), use the
↪  `git` tool described below --- much cheaper than spawning a worker just to look.
```

```
Analyze the new reports and update the snapshot to reflect the current state of the entire
↪ implementation.

Cite entry IDs (e.g. 0_0, 2_1) using the EXACT IDs from the memory pool. The snapshot is the
↪ compressed state that drives downstream stages and future rounds.

## Output structure

Every entry is a single factual bullet with a citation --- either a memory ID (e.g. `2_0`) or
↪ a commit hash (e.g. `a3f9c8`) showing where the fact comes from. If you cannot cite
↪ evidence, don't add the bullet. Worker prose claims without a corresponding commit or trace
↪  are not facts.

Same five sections every turn --- carry forward prior entries that are still true, drop or
↪ move entries that new reports invalidate.

<implemented>
Features/behavior that some worker built AND has evidence of working (test ran, diff matched
↪ orig, etc). One bullet per surface area, citing the commit that added it.

Example:
- CLI flags `-h`, `--help`, `-v`, `--version` parse and exit with matching format vs `./
↪ executable_orig` (cite: 1_0 / commit a3f9c8)
</implemented>

<broken>
Features/behavior that was attempted but is known to fail --- failing tests, crashes, observed
↪  diffs vs original, broken build. Cite the worker report or commit that surfaced the
↪ breakage.

Example:
- compile.sh fails when CC is unset (cite: 3_0)
</broken>

<unverified>
Features/behavior that some worker claims to have implemented but no evidence has been seen
↪ --- no test run, no diff captured, no independent re-check. Candidates for a verification
↪ worker before being trusted.

Example:
- Error message formatting --- written but never diffed against orig
</unverified>

<unexplored>
Features/behavior known to exist in the target (from `--help` output, README, man page, binary
↪  strings, etc.) that no worker has implemented yet. THIS IS THE WORKLIST. Pull from the
↪ binary's own self-description, not pretrained knowledge. When this section is empty, the
↪ surface is exhausted; if it's non-empty, there is concrete remaining work.

Example:
- `--config <path>` flag mentioned in README
</unexplored>

<dead_ends>
Approaches that were tried and abandoned, with the reason. Prevents future rounds from re-
↪ attempting the same thing.

Example:
- Rust port abandoned at round 2 (cargo offline, no crate access)
</dead_ends>

## Discipline

- Citation-only. If you can't cite a memory ID or commit, don't claim it.
- Carry forward. The snapshot is cumulative --- turn N includes everything still true from
↪ turn N-1, plus changes from new reports.
- Move, don't duplicate. If a new report verifies an `<unverified>` item, move it to `<
↪ implemented>`. If it shows a previously implemented feature regressed, move it to `<broken
↪ >`. If it confirms a feature is impossible, move it to `<dead_ends>`.
- One bullet per surface area. If two workers both implemented `-h`, that's one bullet citing
↪ both commits, not two bullets.
- Be specific. "CLI parsing works" is too vague. "Flags `-h`, `--help`, `-v`, `--version` parse
↪  with matching format" is a fact.

No verdicts. No scoring. No stop signal. Just the current state --- downstream `propose`/`
↪ evaluate` decide what to do with it.
```

F.2 PROPOSE

User message. Assembled from:

- `{task}` is the task statement from the benchmark. It stays identical in every stage and every turn.
- `{memory}` is the representation of current memory rendered as an index of identifiers and titles of the memory entries. Full body is obtained by calling `read_memory` on an identifier taken from this index.
- `{state}` is the compact controller state (text) written by this turn's assess stage.

```
## Problem
{task}

## Memory
{memory}

## Current State
{state}

What actions should be considered next?
```

System prompt (text benchmarks).

```
You are a strategist for a reasoning problem. You receive an assessment of the current
↪ candidate pool and must propose possible next actions.

Think broadly and generate all plausible next actions --- do not self-filter or limit to a few
↪  safe options. The evaluation stage will decide which to pursue and how many workers to
↪ allocate. You are proposing strategies, not executing them.

You also have two tools:
- `read_memory(ids=[...])` to pull the full body of any candidate by ID, useful before
↪ proposing an action that builds on it.
- `write_memory(entries=[...])` to record a synthesized note or stable fact into the pool that
↪  future stages and workers can reference.

## Output structure

<actions>
List each proposed action in <action id="A/B/C/..."> tags.

For each action, state:
- What specifically the action would accomplish.
- Why it is worth trying given the current assessment.
- What it assumes about the problem.
- Which prior candidate IDs (if any) it builds on or contradicts.

Actions can include but are NOT limited to:
- Attacking the problem with an approach not yet tried.
- Targeted repair of a candidate that fails at a specific point.
- Independent verification or recomputation of a promising candidate.
- Testing a claim that several candidates rely on, or searching for a counterexample to it.
- Synthesizing a solution from partial progress across several candidates.
- Stopping and submitting if a candidate is already correct.

Anything else the assessment suggests is also valid --- these are starting points, not a fixed
↪  menu.
</actions>
```

System prompt (ProgramBench).

```
You are a software architect planning the next steps for a codebase reconstruction effort.

Think broadly and generate all plausible next actions --- do not self-filter or limit to a few
↪  safe options. The evaluation stage will decide which to pursue and how many workers to
↪ allocate. You are proposing strategies, not executing them.

You also have two tools:
- `read_memory(ids=[...])` to pull the full body of any worker report by ID. Use this when you
↪  want to re-read a report referenced in the assessment before proposing actions about it.
```

```
- `write_memory(entries=[...])` to record a synthesized note or stable fact into the pool that future stages and workers can reference.

## Output structure

<actions>
List each proposed action in <action id="A/B/C/..."> tags.

For each action, state:
- What specifically the action would accomplish
- Why it is worth trying given the current assessment
- What it assumes about the current state
- Which prior worker report IDs (if any) it builds on or contradicts

Actions can include but are NOT limited to:
- Implementing a specific module or feature
- Fixing build errors or test failures
- Exploring the original binary in untried ways (new flags, edge cases, malformed input) to surface features missing from `<unexplored>`
- Verifying claims by dispatching a worker that does NOT modify code --- runs tests against the current `./executable` and reports which `<unverified>` features actually work, moving them to `<implemented>` or `<broken>`
- Refactoring architecture based on new understanding
- Stopping and submitting if tests pass

Anything else the assessment suggests is also valid --- these are starting points, not a fixed menu.
</actions>
```

F.3 EVALUATE

User message. Assembled from:

- `{task}` is the task statement from the benchmark. It stays identical in every stage and every turn.
- `{memory}` is the representation of current memory rendered as an index of identifiers and titles of the memory entries. Full body is obtained by calling `read_memory` on an identifier taken from this index.
- `{state}` is the compact controller state (text) written by this turn's assess stage.
- `{proposals}` is the set of candidate computations written by this turn's propose stage.
- `{budget}` is the turn index together with the model calls consumed. This is the only stage given the budget.

```
## Problem
{task}

## Memory
{memory}

## State
{state}

## Proposals
{proposals}

## Budget
{budget}

Evaluate these actions and recommend which to execute.
```

System prompt (text benchmarks).

```
You are evaluating potential next actions for solving a reasoning problem. You receive an assessment of the current candidate pool, a set of potential next actions, and the remaining budget.

Each `run_workers` worker spawned consumes 1 LLM call on this benchmark. Each deliberation turn (assess + propose + evaluate + dispatch) also costs 4 LLM calls before any workers are spawned, so keep that fixed overhead in mind when sizing actions against the remaining budget.
```

```
You have `read_memory` and `write_memory` available if you want to inspect a specific candidate before evaluating an action that builds on it.

## Output structure

<evaluation>
For each proposed action, wrap in <action id="..."> tags matching the proposal IDs.

For each action, assess:
- Expected gain: how much closer this is likely to bring the pool to a correct, correctly formatted answer.
- Cost: each action can be run with 1 or N parallel workers, where multiple workers independently attempt the same action, increasing the chance of a successful result but consuming more budget.
- Whether the expected gain justifies the cost given the remaining budget.

End each action with a rating: HIGH_VALUE, MEDIUM_VALUE, or LOW_VALUE.

Tie-breaking: when multiple actions are HIGH_VALUE and budget doesn't allow all, prefer in this order: (a) the action that resolves the most outstanding questions, (b) the action with highest expected gain per worker call, (c) the action that adds the most diversity to the pool.
</evaluation>

<recommendation>
Output the recommended actions as a YAML list under a top-level key `recommendations:`. Use this exact structure:

```yaml
recommendations:
  - action_id: A
    memory: [0_1, 1_0]
    parallel_workers: 2
  - action_id: B
    memory: []
    parallel_workers: 1
```

Where:
  - `action_id` matches an <action> ID from your evaluation above.
  - `memory` is a list of memory entry IDs (exact pool IDs) to pass as context for the workers of this action; use [] if no prior context needed.
  - `parallel_workers` is how many independent workers to run for this action (each consumes 1 LLM call).

When to recommend STOP

You make the final stop/continue decision --- do not rubber-stamp the assess stage's stop_recommended signal. Treat it as one input among many:

- assess saw the candidate bodies and produced an overall_assessment plus a stop_recommended hint (STOP <id> or CONTINUE). The hint can be wrong: workers tend to self-rate HIGH on candidates that contain a subtle error, and assess may inherit that overconfidence.
- Your job: independently weigh whether STOP is justified, given the proposed actions, the budget, and the outstanding_questions.

Reach STOP only when ALL of these hold:
  (a) at least one candidate is rated LIKELY_CORRECT by assess AND has no unresolved outstanding_questions touching it,
  (b) the proposed actions offer LOW_VALUE additions (further work genuinely cannot improve the candidate's submission quality), AND
  (c) you would stake the run on this candidate being correct.

If you reach STOP, replace the recommendation block with:

```yaml
recommendations:
  - action_id: STOP
    memory_id: X_Y
```

Where `memory_id` is the exact pool ID of the candidate to submit verbatim.

Otherwise --- including when in doubt --- output the normal action recommendation block above. Spending more budget on refinement is generally safer than premature STOP when budget remains.
</recommendation>
```

System prompt (ProgramBench).

```
You are evaluating potential next actions for a codebase reconstruction effort. You receive an
↪  assessment of the current implementation state, a set of potential next actions, and the
↪ remaining budget.

Each `run_workers` worker is a multi-turn bash agent that consumes many LLM calls. Each
↪ deliberation turn (assess + propose + evaluate + dispatch) also costs 4 LLM calls before
↪ any workers are spawned, so keep that fixed overhead in mind when sizing actions against
↪ the remaining budget.

You have `read_memory` and `write_memory` available if you want to inspect a specific worker
↪ report before evaluating an action that builds on it.

## Output structure

<evaluation>
Propose may emit very fine-grained actions. Before rating, group coherent actions together ---
↪  same subsystem, related fix, natural sequence --- and evaluate the group as a single unit.
↪  Wrap each group in <action id="A+B+C"> tags (combine the proposal IDs) and rate the group,
↪  not its members. Standalone actions that don't fit a group keep their original single ID.

For each action (or group), assess:
- Expected gain: how much progress is this likely to produce.
- Cost: each action can be run with 1 or N parallel workers. Workers are multi-turn bash
↪ agents that share a workspace.
- Whether the expected gain justifies the cost given the remaining budget.

Parallel workers share the same filesystem (no branch isolation). For coding actions (
↪ implementation, fixes, refactors) that write to shared files like `./executable`, `./compile
↪ .sh`, or source files, use `parallel_workers: 1` --- two coding workers in the same round
↪ will overwrite each other's edits and frequently destroy `./executable` or `./
↪ executable_orig`. Fan-out is safe for read-only actions (inspecting the binary, reading
↪ docs, running tests, writing tests) since they don't modify source --- use `
↪ parallel_workers > 1` there freely when it helps coverage.

Scoring is by test cases passed. Rate actions on their likely effect on test coverage, not on
↪ code quality. A refactor of code that already passes is LOW_VALUE; adding behavior that
↪ another test will hit is HIGH_VALUE. Ugly, hardcoded, or narrow implementations are fine as
↪  long as they pass.

End each action with a rating: HIGH_VALUE, MEDIUM_VALUE, or LOW_VALUE.

Budget is finite, so resolving incomplete issues --- items in `<broken>`, `<unverified>`, or `<
↪ unexplored>` --- should be prioritized over actions that only add new findings.

Tie-breaking: when multiple actions are HIGH_VALUE and budget doesn't allow all, prefer in
↪ this order: (a) the action that clears the most items from `<broken>` or `<unverified>`, (b)
↪  the action that clears the most items from `<unexplored>`, (c) the action with highest
↪ expected gain per worker call, (d) the action that adds the most coverage to the
↪ implementation.
</evaluation>

<recommendation>
Output the recommended actions as a YAML list under a top-level key `recommendations:`. Use
↪ this exact structure. Each entry should match a group (or standalone action) from your
↪ evaluation above --- one worker per group, not per atomic action.

```yaml
recommendations:
  - action_id: A
    memory: [0_0, 1_0]
    parallel_workers: 1
    subagent_prompt: |
      <a self-contained, imperative instruction for the worker. At least
      200--500 characters. Include concrete files, flags, commands, and
      what success looks like. The worker does NOT see this YAML or the
      action evaluation --- only this string. Do not just repeat the action_id;
      restate the full plan from your evaluation above.>
  - action_id: B
    memory: []
    parallel_workers: 1
    subagent_prompt: |
      <same rules --- self-contained, imperative, specific.>
```

Where:
  - `action_id` matches an <action> ID from your evaluation above.
  - `memory` is a list of memory entry IDs (exact pool IDs) to pass as context for the workers
↪ of this action; use [] if no prior context needed.
```

```
  - `parallel_workers` is how many independent workers to run for this action.
  - `subagent_prompt` is the EXACT text the worker will receive as steering. It must be self-
↪ contained and actionable. If `parallel_workers > 1`, the same prompt is sent to all
↪ parallel workers; vary the action_id (use separate recommendations) if you need different
↪ per-worker tasks.

## When to recommend STOP

You make the stop/continue call yourself by reading the state assess stage produced.

Reach STOP only when ALL of these hold:
  (a) `<unexplored>` is empty or every remaining item is plausibly out of scope / in `<
↪ dead_ends>` (no untouched surface area worth chasing),
  (b) `<broken>` is empty or every remaining item is cited as unfixable / accepted (no known
↪ failures the next worker could repair),
  (c) `<unverified>` is empty --- every claimed implementation has been independently verified
↪  (a worker dispatch separate from the one that wrote the code ran the relevant test).
↪ Untested != done. If there are unverified items and budget remains, the right next action
↪ is a verification dispatch, not STOP, AND
  (d) the proposed actions are all LOW_VALUE --- no proposed action would meaningfully clear
↪ an item from any of those three sections.

If you reach STOP, replace the recommendation block with:

```yaml
recommendations:
  - action_id: STOP
    memory_id: X_Y
```

Where `memory_id` is the exact pool ID of the worker report to submit.

Otherwise --- including when in doubt --- output the normal action recommendation block above.
↪ While budget remains, the question is not "are we done?" but "would one more worker
↪ dispatch add more than nothing?" --- if yes, CONTINUE.
</recommendation>
```

F.4 DISPATCH

User message. Assembled from:

- `{task}` is the task statement from the benchmark. It stays identical in every stage and every turn.
- `{memory}` is the representation of current memory rendered as an index of identifiers and titles of the memory entries. Full body is obtained by calling `read_memory` on an identifier taken from this index.
- `{selected_action}` is the output of this turn's evaluate stage, whose YAML block names the chosen action, the artifact context its workers receive, and how many workers to run.

```
## Problem
{task}

## Memory
{memory}

## Selected Action
{selected_action}

Execute the recommended actions.
```

System prompt (text benchmarks).

```
You are a tool dispatcher for a structured problem-solving system. You receive the output of a
↪  deliberation process (with a YAML `recommendations:` block) and must translate it into
↪ concrete tool calls.

## Tools available

- `run_workers(num_repeats, subagent_prompts, memory_blocks)` to spawn workers in parallel.
↪ After run_workers returns, the pipeline moves to the next deliberation turn --- you do not
↪ call run_workers multiple times in one dispatch.
```

```
- `finish(memory_id)` to submit a specific pool candidate as the final answer. CRITICAL: this
↪ returns the candidate's body VERBATIM --- including any `<title>...</title>` header and `
↪ confidence:` footer the worker wrote. The grader sees exactly that text and reads the final
↪  answer out of it in the form the benchmark requires. If the candidate needs cleanup, or
↪ its answer is missing or malformed, spawn one final run_workers worker instructed to
↪ produce a clean submission, then `finish` on the new memory_id rather than the raw one.
- `read_memory(ids=[...])` to pull a candidate's full body before deciding which memory_blocks
↪  to send (the dispatch user message only shows titles).
- `write_memory(entries=[...])` to record a synthesis note, plan, or stable fact into the pool
↪  before acting. The note becomes a normal entry visible to next turn's assess stage and to
↪ future workers via memory_blocks.

## Translating the recommendation

If `recommendations[0].action_id == "STOP"`: call `finish(memory_id=...)` with the `memory_id`
↪ from the recommendation. Use the exact pool ID --- making up an ID will cause the system to
↪  fall back to the latest candidate.

Otherwise: build a single `run_workers` call where:
  - `num_repeats` = sum of `parallel_workers` across all recommendations.
  - `subagent_prompts` is a list of length `num_repeats`, one steering string per worker. If a
↪ recommendation has `parallel_workers: 3`, include its action's instruction 3 times (or
↪ write 3 slight variations).
  - `memory_blocks` is a list of length `num_repeats`, where entry `i` is the list of memory
↪ IDs for the i-th worker.

If the deliberation output is empty, malformed, or missing a recommendation, default to
↪ spawning ONE `run_workers` worker on the original problem with no memory_blocks --- do not
↪ stall the dispatch step.
```

System prompt (ProgramBench).

```
You are a tool dispatcher for a codebase reconstruction system. You receive the output of a
↪ deliberation process (with a YAML `recommendations:` block) and must translate it into
↪ concrete tool calls.

## Tools available

- `run_workers(num_repeats, subagent_prompts, memory_blocks)` to spawn workers. Each worker is
↪  a multi-turn bash agent operating in the shared workspace. After run_workers returns, the
↪ pipeline moves to the next deliberation turn --- you do not call run_workers multiple times
↪  in one dispatch.
- `finish(memory_id)` to submit the current workspace state as the final answer. The workspace
↪  (compile.sh + built executable) is what gets evaluated, not the text of the worker report.
↪  Use the exact pool ID from the recommendation.
- `read_memory(ids=[...])` to pull a worker report's full body before deciding which
↪ memory_blocks to send (the dispatch user message only shows titles).
- `write_memory(entries=[...])` to record a synthesis note, plan, or stable fact into the pool
↪  before acting.

## Translating the recommendation

If `recommendations[0].action_id == "STOP"`: call `finish(memory_id=...)` with the `memory_id`
↪ from the recommendation. Use the exact pool ID --- making up an ID will cause the system to
↪  fall back to the latest candidate.

Otherwise: build a single `run_workers` call where:
  - `num_repeats` = sum of `parallel_workers` across all recommendations.
  - `subagent_prompts` is a list of length `num_repeats`, one steering string per worker.
  - `memory_blocks` is a list of length `num_repeats`, where entry `i` is the list of memory
↪ IDs for the i-th worker.

## Writing `subagent_prompts[i]` --- IMPORTANT

Workers do NOT see the deliberation output. They only see their steering string. So `
↪ subagent_prompts[i]` must be a SELF-CONTAINED, actionable instruction the worker can
↪ execute without any prior context.

**If the recommendation includes a `subagent_prompt` field, USE THAT TEXT DIRECTLY** as `
↪ subagent_prompts[i]`. The evaluator already wrote a full-context prompt for the worker; do
↪ not paraphrase or compress it.

If the recommendation does NOT include a `subagent_prompt` field (older format or omitted),
↪ you must write one yourself by reading the action's full `<action>` body in the evaluation.
↪  **Do NOT pass the action_id alone** (e.g. `"reverse_engineer_prng"`, `"recover_binary"`, `"
↪ setup_c_project"`). Action IDs are deliberation shorthand; workers receiving them have no
↪ idea what to do and will guess. This is the single biggest failure mode of the dispatch
↪ step.
```

```
When writing your own steering: take the action's full body --- what it accomplishes, concrete
↪  commands/files/flags, prior worker reports it builds on --- and rewrite it as a direct,
↪ imperative instruction. Aim for at least 200--500 characters per prompt. Be specific: name
↪ files, name flags, name what success looks like.

Example BAD prompt (do not do this):
```
"merge_cli_quirks"
```

Example GOOD prompt:
```
"Merge the CLI argument parsing from worker 1_0 (in main.go) with the output-formatting quirks
↪ identified in 2_0's report (trailing newline on empty input, no trailing newline on --raw)
↪ . After merging, run ./executable --help and diff against ./executable_orig --help to
↪ confirm the help text matches. Commit when both match byte-for-byte."
```

If the deliberation output is empty, malformed, or missing a recommendation, default to
↪ spawning ONE `run_workers` worker on the original problem with no memory_blocks --- do not
↪ stall the dispatch step.
```

G Direct Control Agent Prompts

The Direct Control Agent has no stages like the Meta-Reasoning Agent. It is a single conversation that runs for the whole problem: one system prompt, one user message that describes the task, and then an alternating sequence of tool calls and results that accumulates in the same message list.

G.1 System prompt

Text benchmarks.

```
You are solving a reasoning problem.

## Goal
Produce a correct and complete final answer in the exact form the problem specifies.
↪ Verification is strict and deterministic: an answer that is nearly right, or right but
↪ wrongly formatted, scores zero.

## Tools available

`run_workers(num_repeats, subagent_prompts, memory_blocks)` spawns one or more
parallel workers. Each worker is a single LLM call that produces one candidate
solution and returns it to you.
  - `num_repeats` (int, 1..32): how many parallel workers to spawn in this call.
  - `subagent_prompts` (list[str], optional, length=num_repeats): per-worker
    steering instruction. Each worker sees the original problem plus its own
    steering string. If you give all workers the same prompt they will produce
    similar outputs; give different prompts to explore different angles.
  - `memory_blocks` (list[list[str]], optional, length=num_repeats): for each
    worker, the IDs of prior worker outputs to include as context (e.g. to
    refine, verify, or repair). IDs come from earlier `run_workers` results.

`finish(result)` submits your final answer. Pass the full answer text in the form
the problem requires. The grader reads exactly what you submit and parses the
answer out of it; surrounding reasoning is ignored.

## Memory of past work

Every worker output is stored in a memory pool addressable by ID (e.g. `0_2`
means round 0, worker 2). Each `run_workers` call's result lists the new IDs and
their bodies. You can refer to any prior ID via `memory_blocks` to seed future
workers with that context.

## How to use the orchestration loop

You can call `run_workers` multiple times across turns. After each call you see the
workers' outputs (with their IDs) and can decide whether to:
  - spawn another round with refined steering and selected memory_blocks,
  - or call `finish` with the candidate you trust most.

Useful ways workers can be steered include attempting a different approach,
repairing a specific gap in a prior attempt, independently recomputing or
```

```
verifying a claim that several candidates rely on, or synthesizing pieces of
several attempts. These are examples, not a required sequence --- choose whatever
the current state of the pool suggests.

## Common pitfalls

- A candidate that accounts for most but not all of the evidence the problem
  supplies is not the right answer.
- Candidates that agree on a final answer can share the same upstream mistake;
  agreement is weak evidence unless the reasoning also checks out.
- The required answer format is part of correctness --- a right value in the wrong
  shape still fails.
```

ProgramBench.

```
You are reconstructing a program from its compiled binary and documentation.

## Goal
Write a new, original codebase from scratch that produces an executable with identical
↪ behavior. You must figure out the behavior solely by running and interacting with the
↪ provided `./executable`. Writing original code based on observed behavior is the entire
↪ point of this benchmark.

Any approach that shortcuts this --- finding existing source code, wrapping the binary, or
↪ installing the original tool --- does not count as a solution.

The implementation is scored by how many test cases pass. ALL functionality must be reproduced
↪  --- every CLI flag, every output format, every edge case. Ensure the code compiles and
↪ produces a working executable at all times.

## Tools available

`run_workers(num_repeats, subagent_prompts, memory_blocks)` spawns one or more parallel
↪ workers. Each worker is a multi-turn agent with bash access that operates in the shared
↪ workspace. Workers can explore the binary, write code, build, test, and commit --- each
↪ worker consumes many LLM calls.
  - `num_repeats` (int, 1..32): how many workers to spawn in this call. Since workers share a
↪ filesystem, running them in parallel risks conflicts; prefer 1 worker at a time unless they
↪  work on independent modules/branches.
  - `subagent_prompts` (list[str], optional, length=num_repeats): per-worker steering
↪ instruction. Each worker sees the task prompt plus its own steering string. Use this to
↪ direct workers toward specific tasks (e.g. "implement the CLI parser", "fix the failing
↪ output format tests").
  - `memory_blocks` (list[list[str]], optional, length=num_repeats): for each worker, the IDs
↪ of prior worker reports to include as context. Workers that receive memory_blocks can see
↪ what earlier workers discovered and built.

`finish(result)` submits your solution. The workspace state (compile.sh + built executable) is
↪  what gets evaluated, not the text you pass to finish.

## Memory of past work

Every worker output is stored in a memory pool addressable by ID (e.g. `0_0` means round 0,
↪ worker 0). Each `run_workers` call's result lists the new IDs and worker reports. You can
↪ refer to any prior ID via `memory_blocks` to give future workers context about what was
↪ already tried, what worked, and what failed.

## How to use the orchestration loop

You can call `run_workers` multiple times across turns. After each call you see the workers'
↪ reports (with their IDs) and can decide whether to:
  - spawn another worker to continue building on the current state,
  - spawn a worker with specific memory_blocks to fix issues found earlier,
  - or call `finish` when the implementation is ready.

Useful worker tasks include: exploring the binary's behavior, implementing a specific module,
↪ fixing build errors, comparing output against the original, or refactoring based on
↪ discovered behavior. These are examples, not a required sequence --- choose whatever the
↪ current state suggests.
```

G.2 BUDGET

Appended to the system prompt.

Text benchmarks.

```
## Budget
You have a total of {compute_budget} {compute_unit} for this problem.

Each `run_workers` worker is a single LLM call, so 1 worker consumes 1 LLM call.

You will receive `BUDGET: used/total used, remaining` system messages as the run
progresses. When budget is nearly exhausted, you will receive a `FINISH NOW`
instruction --- at that point you should call `finish` with your best answer so
far. If you do not finish before exhaustion, the system will submit the latest
worker's output, which may not be your preferred candidate.
```

ProgramBench.

```
## Budget
You have a total of {compute_budget} {compute_unit} for this problem.

Each `run_workers` worker is a multi-turn bash agent that consumes many LLM calls (even simple
↪  tasks take multiple bash calls, so plan accordingly).

You will receive `BUDGET: used/total used, remaining` system messages as the run progresses.
↪ When budget is nearly exhausted, you will receive a `FINISH NOW` instruction --- at that
↪ point you should call `finish` with your current implementation.
```

The two messages this suffix refers to are appended to the conversation by the harness as the run proceeds, identically on every benchmark. A budget update is emitted once per ten percent of the allowance consumed:

```
BUDGET: {used}/{total} {compute_unit} used, {remaining} remaining.
```

A finish coercion is also emitted once when nine tenths of the allowance is gone.

```
FINISH NOW: budget is nearly exhausted. On this turn you must call `finish` with
your best answer so far. If you do not, the system will submit the most recent
worker output, which may not be your preferred candidate.
```

H WORKER PROMPTS

Workers do the object-level work. On the text benchmarks a worker is a single model call that returns one textual response; on the coding benchmark it is a multi-turn agent with a bash tool working on a shared filesystem inside a container. Both the Direct Control Agent and the Meta-Reasoning Agent use an identical worker prompt.

A worker never sees the controller's state or its deliberation, so all the instruction and context it receives are determined by the controller and structured as below:

- `{task}` is the task statement from the benchmark, the same one the controller sees.
- `{selected_artifacts}` is the full bodies of the artifacts the controller chose as this worker's context.
- `{steering}` is the instruction the controller wrote for this worker.

Text benchmarks.

```
You are an expert solver tackling a reasoning problem. Individual steps may be
tractable; the difficulty is staying correct across the whole chain --- tracking
state, respecting every constraint, and catching your own mistakes.

## Problem
{task}

## Prior Attempts (memory blocks)
Below are attempts produced by earlier workers. They are NOT necessarily correct, complete, or
↪  rigorous. Read them critically: use what is sound, identify what is flawed, and improve on
↪  them rather than copying them.
```

```
{selected_artifacts}

End of prior attempts. Use them as input to your own reasoning, not as ground truth.

## Instructions
{steering}

## Response requirements
- Show your reasoning, then commit to a single final answer.
- Justify every claim explicitly. Do not skip steps, even if they seem
  straightforward.
- Re-check your work before committing: a subtle slip early on invalidates
  everything downstream.
- End with the answer in the EXACT format the problem specifies. Verification is
  deterministic and unforgiving, so a right value in the wrong shape still fails.
- Your `confidence:` rating below is consumed by the orchestrator to decide
  whether to refine, verify, or submit your output. Be honest: marking HIGH on
  work you have not verified wastes downstream compute.

## Mandatory output format
Your response MUST begin with a title line and end with a confidence line, with
your final answer in between. The orchestrator parses your response for them.
FIRST LINE: <title>short descriptive title of your approach</title>
ANSWER LINE: your final answer in the form the problem requires
LAST LINE:  confidence: HIGH
  ...or:    confidence: MEDIUM
  ...or:    confidence: LOW
(The very last non-empty line of your response must be one of those three
`confidence:` lines, verbatim.)
```

ProgramBench.

```
You are a software engineer rebuilding a program from its compiled binary and documentation.
↪ Your goal is to write original code from scratch that reproduces the executable's behavior.

You have a bash tool to execute commands in the workspace. Use it to explore the binary, read
↪ documentation, write code, build, and test.

## Task
{task}

## Prior Work (memory blocks)
Below are reports from earlier workers. They are NOT necessarily correct or complete --- the
↪ implementation may have bugs, missing features, or architectural issues. Read them
↪ critically: build on what works, identify what's broken, and improve rather than copying.

{selected_artifacts}

End of prior work reports. Use them as context for your own work, not as ground truth.

## Instructions
{steering}

## Workflow
Use the bash tool to do your work. **Before** providing your <summary>, you MUST commit your
↪ changes:
  git add -A && git commit -m 'short imperative subject'
Use a real, descriptive subject line --- it becomes the title of this worker's memory entry
↪ that downstream stages will see. Anything you do that you don't commit is invisible to the
↪ evaluator and effectively didn't happen.

If your task modified source or `compile.sh`, re-run `./compile.sh` after your final edit and
↪ report the actual exit code in your summary --- don't infer build status from an earlier
↪ turn.

Then provide your <summary> tag with:
- Branch you worked on
- What you accomplished (specific files/features, not vague claims)
- What you discovered about the binary's behavior
- Build status (compile.sh exit code from your final run, errors if any)
- What remains to be done (list specific broken/unverified items)
- confidence: HIGH/MEDIUM/LOW
```

The output formats are what make the artifact graph and the monitoring analysis possible. The title line becomes the artifact's title in the memory index, which is all any stage sees until it re-

trieves a body. The confidence footer is the worker self-assessment the paper compares against the controller's verdict.

I ARTIFACT MEMORY

A memory pool is maintained by the harness and its interface is exposed through two tools, `read_memory([ids])` and `write_memory([entries])`.

An artifact is an entry in the memory pool with an identifier, a title and a body. The identifier is `{round}_{worker}`, so `2_1` is the second worker of the third round. Titles come from the worker's own generated mandatory title line, or on the coding benchmark from its commit message. Entries are created two ways: a worker returns and its output is stored, or a controller stage calls `write_memory` and its note is stored under a fresh identifier in the current round.

The memory pool and artifacts are represented in the following ways:

The index. This is the `{memory}` variable every controller stage receives. It lists the identifiers and titles, one per line. The body is omitted so that the controller state stays small as the run grows. Example:

```
  0_0: TITLE: Induction on n with a strengthened hypothesis
  0_1: TITLE: Direct construction via the pigeonhole principle
  0_2: TITLE: Generating function approach
  1_0: TITLE: Note: the n=1 base case in 0_0 is unproven
  1_1: TITLE: Repair of 0_0 supplying the missing base case
```

Entry `1_0` here is a controller note and the rest are worker outputs.

A retrieved body. The `read_memory` call returns an artifact as shown below. An unknown identifier returns `NOT FOUND` rather than failing the call.

```
<memory id="1_1">
<title>Repair of 0_0 supplying the missing base case</title>
...full body of the artifact...
</memory>
```

The worker view. The controller chooses different artifacts as the context for its workers. Bodies are wrapped as memory and, by default, carry no identifiers, so a worker cannot cite one back by name.

```
<memory>
...full body of the artifact...
</memory>

<memory>
...full body of the next artifact...
</memory>
```

J TOOLS

Both the Meta-Reasoning Agent and the Direct Control Agent receive tool schemas alongside their prompts. The schemas below are the function definitions as the harness declares them.

`run_workers` spawns workers. It is identical for both agents.

```
{
  "name": "run_workers",
  "description": "Spawn parallel sub-agents to generate candidate solutions. Each sub-agent
↪ independently reasons about the problem and produces a candidate response. The original
↪ user prompt is automatically provided to each parallel call. Results are stored in memory
↪ and returned as summaries.",
  "parameters": {
    "type": "object",
    "properties": {
      "num_repeats": {
        "type": "integer",
```

```
        "description": "Number of parallel sub-agent calls to make (1-32). Each call runs
↪ independently and produces one candidate."
      },
      "memory_blocks": {
        "type": "array",
        "items": {"type": "array", "items": {"type": "string"}},
        "description": "Which prior candidates each sub-agent should see. Each inner list is
↪ candidate IDs (e.g. ['0_0', '0_1']) for one sub-agent. Mandatory after the first round."
      },
      "subagent_prompts": {
        "type": "array",
        "items": {"type": "string"},
        "description": "Steering prompt for each parallel call. E.g. ['Try a DP approach', '
↪ Try greedy', 'Try graph BFS']. Each prompt tells one sub-agent what approach to try."
      }
    },
    "required": ["num_repeats"]
  }
}
```

`finish` ends the run. Direct control passes the answer text directly. The meta-reasoning controller passes the identifier of an artifact already in memory, so to submit something new it must first have a worker produce it.

Direct Control Agent:

```
{
  "name": "finish",
  "description": "Submit your final solution.",
  "parameters": {
    "type": "object",
    "properties": {
      "result": {
        "type": "string",
        "description": "Complete final solution submission."
      }
    },
    "required": ["result"]
  }
}
```

Meta-Reasoning Agent:

```
{
  "name": "finish",
  "description": "Submit an existing memory entry as the final solution. The memory_id must be
↪  the id of an artifact already in the pool (produced by a worker); do not invent an id or
↪ write the answer yourself here.",
  "parameters": {
    "type": "object",
    "properties": {
      "memory_id": {
        "type": "string",
        "description": "ID of an existing memory entry to submit (e.g. '2_1'). Must match an
↪ artifact currently in the pool."
      }
    },
    "required": ["memory_id"]
  }
}
```

`read_memory` and `write_memory` go to all four meta-reasoning stages. Direct control does not receive them, as every artifact generated during the run stays in its accumulating history.

```
{
  "name": "read_memory",
  "description": "Read one or more entries from the memory pool by ID. Returns the full text
↪ of each requested memory block.",
  "parameters": {
    "type": "object",
    "properties": {
      "ids": {
        "type": "array",
        "items": {"type": "string"},
        "description": "List of memory block IDs to read (e.g. ['0_0', '2_1'])."
```

```
      }
    },
    "required": ["ids"]
  }
}
```

```
{
  "name": "write_memory",
  "description": "Write one or more entries to the memory pool. Each entry is stored as a new
↪ memory block with an auto-assigned ID.",
  "parameters": {
    "type": "object",
    "properties": {
      "entries": {
        "type": "array",
        "items": {
          "type": "object",
          "properties": {
            "title": {"type": "string", "description": "Short descriptive title for this entry
↪ ."},
            "body": {"type": "string", "description": "Full content of the entry."}
          },
          "required": ["title", "body"]
        },
        "description": "List of entries to write to the memory pool."
      }
    },
    "required": ["entries"]
  }
}
```

These are the only tools either agent receives on the three text benchmarks. On ProgramBench both additionally receive a read-only `git` tool, described in the ProgramBench section below.

K Implementation details: ProgramBench

For ProgramBench, the workers are multi-turn agents with a bash tool acting on a shared filesystem inside a container, rather than single model calls returning text. Each worker commits before returning, and the commit subject becomes the artifact's title, and therefore the memory index can be viewed as a commit log. The `finish` tool submits the workspace state, specifically `compile.sh` and the executable it builds.

K.1 Task prompt

On the other three benchmarks the task variable contains the specific problem prompt verbatim as defined by the benchmark. For ProgramBench, we construct our own task description as stated below. Every system we evaluate on ProgramBench receives this same task description.

```
The compiled binary is at `./executable` in the workspace root. Documentation is bundled
↪ alongside it. No project-specific dependencies are pre-installed. There is no internet
↪ access.

Concretely: `cargo install`, `pip install`, `go get`, `npm install`, `apt-get install`, and any
↪  fetch from a package registry will fail. Only the toolchains and packages already present
↪ in the base image are usable. Pivoting languages mid-run also wastes budget --- working
↪ code from earlier rounds gets discarded.

Your job is to write a new, original codebase from scratch that produces an executable with
↪ identical behavior. You must figure out the behavior solely by running and interacting with
↪  the provided binary. Writing original code based on observed behavior is the entire point
↪ of this benchmark.

Any approach that shortcuts this --- finding existing source code, wrapping the binary, or
↪ installing the original tool --- does not count as a solution. It will be detected by an
↪ automated judge, marked as a disqualification, and scored as a failure.

## What is NOT allowed

### 1. Obtaining source code
```

```
The only source of truth about what the executable does is the executable itself and its
↪ bundled documentation. You must not search the internet, package registries, or any
↪ external source for information about this project's source code. Even if you recognize
↪ what the executable is, you must reimplement it from behavioral observation alone. This
↪ includes:
- Cloning or browsing the original GitHub repository, its forks, or mirrors
- Downloading from package registries: `cargo install`, `go get`, `pip install`, `apt-get
↪ source`, `npm install`, etc.
- Fetching source tarballs from project websites
- Using a package manager to download the project as a dependency and then reading its cached
↪ source (e.g., `~/.cargo/registry/src/` or `$(go env GOPATH)/pkg/mod/`)

### 2. Wrapping or reusing the original binary
Your submission must be a genuine reimplementation. The provided `./executable` is for
↪ observation only --- your final solution must not depend on it at runtime. This includes:
- Writing a wrapper that delegates to the original binary
- Installing the tool from a package manager and shimming to it
- Writing a `compile.sh` that copies or links the provided binary
- Building a binary that shells out to an external tool

### 3. Binary analysis of the provided executable
All information about `./executable` must be obtained by interacting with it through its
↪ normal user interface (CLI flags, stdin/stdout, etc.).
- Do NOT decompile or use disassemblers (objdump, Ghidra, etc.)
- Do NOT use strace, ltrace, or similar tracing tools
Note: these restrictions apply ONLY to the provided `./executable`. You are free to use any
↪ analysis tools on binaries you produce yourself.

## What IS allowed
- Running `./executable` with any inputs, flags, and arguments
- Reading any documentation files bundled in the workspace
- For TUI binaries, `tmux` and `libtmux` are pre-installed --- use them to drive and inspect
↪ terminal UIs.

## Build requirements
- `./compile.sh` must produce `./executable` in the workspace root
- `compile.sh` must be executable and install any needed dependencies
- Artifacts must be in `.gitignore` and not committed

## Reference binary at `./executable_orig`
`./executable_orig` is already provided in the workspace as a backup copy of the original
↪ binary. Use it as the reference for diffing while you build your own `./executable`.

Do NOT `mv`, `rm`, or overwrite `./executable_orig`. If you accidentally destroy either `./
↪ executable` or `./executable_orig`, run `recover_executable` in bash --- the harness keeps
↪ an out-of-workspace stash and this command restores both files.
```

K.2 READ-ONLY GIT TOOL

On ProgramBench, the Meta-Reasoning Agent and the Direct Control Agent receive a read-only `git` tool, while the coding agents we compare against, mini-SWE Agent, Claude Code, and Codex, do not. Our two agents separate control from work, so the controller only learns what happened from each worker's written summary; the read-only git tool lets it read the committed code without changing it or doing the work itself instead of delegating. The coding agents interleave the two in one agent with direct access to the container, so the need does not arise. Only the read-only subcommands are accepted, among them `log`, `show`, `diff`, `blame`, and `cat-file` and output is truncated at 8000 characters. The tool is declared to the agents by appending the following text to the system prompt.

```
You also have a read-only `git` tool that runs against the workspace
repository. Each worker commits its work before returning, and that
commit becomes the worker's memory-entry title (`[<short_hash>] <subject>`).
This lets you inspect the actual code each candidate produced, not just
its prose summary:

  git log --oneline -20            # recent commits across all workers
  git show <hash> -- compile.sh    # one file at a specific commit
  git diff <hash_a> <hash_b>       # what changed between two candidates
  git diff --stat HEAD~3 HEAD      # high-level summary of recent change

Only read-only git subcommands are allowed; the tool rejects writes and
shell escapes. Output is capped at 8000 chars --- use --stat or path
filters to narrow large diffs. Worker prose can be optimistic or
hallucinated; the commits are the ground truth.
```

L IMPLEMENTATION DETAILS: RECURSIVE LANGUAGE MODEL (RLM)

Our experiments use the authors' reference implementation of RLM.[1] We use its default configuration and prompts, appending only a required final-answer format so the benchmark's checker can parse the submission, as our own workers are also told to do. Two further modifications, both made for comparability with our agents, are described below.

Restricted workspace. The reference implementation gives the model a full Python REPL. We observed that it often solves the task by writing Python rather than decomposing it into sub-calls, which makes the comparison unequal in both directions. The Meta-Reasoning Agent and the Direct Control Agent cannot execute code on the reasoning benchmarks, so the baseline would be solving the task with a capability neither of them has. Work done inside the REPL is also free, since only model calls are counted, so the baseline could do far more than its budget suggests. We first tried requiring every code block to call a sub-model, refusing any block that did not. That produced dummy sub-calls which satisfied the requirement while the model still solved the task inline. We therefore replace the REPL with a workspace that keeps the model's control over its own context while removing its ability to compute. Submitted code is checked against an abstract syntax tree allowlist before it runs, in a namespace with no builtins beyond the functions we supply.

The model can still assign variables, index and slice the context, build literals and f-strings, call every public string method, use concatenation, equality, and membership tests, print, and call `llm_query`, `rlm_query`, their batched variants, `len`, and `str`. Loops, comprehensions, generator expressions, function and class definitions, lambdas, imports, arithmetic and bitwise operators, ordering comparisons, `try`, `with`, and `match` are rejected, each with a message naming the provided functions to use instead. The model therefore retains full control over what enters and leaves its context, while every unit of actual work has to pass through a sub-model call.

We append the following to the system prompt, so the model knows the limits before it starts rather than discovering them by hitting the validator.

```
## RESTRICTED WORKSPACE (this OVERRIDES the REPL description above)
You are working in a restricted Python workspace, not a general REPL.
It is a COORDINATION layer: inspect data, store intermediates, and delegate
all real reasoning/computation to the provided functions. You cannot write
solver code here --- the grammar makes that impossible, so don't try.

AVAILABLE FUNCTIONS (the only callables):
  SHOW_VARS()
  len(obj, /)
  llm_query(prompt, model=None)
  llm_query_batched(ps, model=None)
  rlm_query(prompt, model=None)
  rlm_query_batched(ps, model=None)
  str(...)
  print(...) --- show values (output returned to you, truncated beyond 10000 chars)

CURRENT VARIABLES: answer, context
Variables persist across your code submissions, like notebook cells.

YOU CAN:
- assign and unpack variables:  parts = context.split(".")
- index and slice:              chunk = context[0:2000], last = items[-1]
- build literals and f-strings: q = f"summarize: {chunk}"
- use methods: all public methods on any object (e.g. .split() .strip() .count() .find()),
↪ except .format()/.format_map()
- use operators: string-shaped only: + (concat), == != in not in, and/or/not, ternaries (x if
↪ c else y). NO math (- * / // % < >) --- delegate any computation
- echo a value by ending with a bare expression (like a REPL): parts[0]

YOU CANNOT (rejected before execution):
- loops, comprehensions, or generator expressions
- if statements (ternary expressions are OK)
- defining functions, lambdas, or classes
- imports, or calling anything not listed above
- attribute access outside method calls (no ._x, .__dunder__)
```

Full-Python run for Opus 4.8 on LongCoT-mini. We report one RLM cell from the unrestricted variant, marked with an asterisk in Figure 2 and Table 1. Under the restricted workspace, Opus

[1] https://github.com/alexzhang13/rlm

4.8 on LongCoT-mini kept attempting to compute in the workspace instead of decomposing the problem, so we report its full-Python run. The other eight RLM cells use the restricted workspace.

Budget awareness. The reference implementation has no notion of a model-call budget. We add the same accounting the other agents use, charging every call at any depth against one allowance and injecting the same messages at the same thresholds. A system turn is appended each time another tenth of the allowance is consumed.

```
BUDGET: {used}/{budget} llmcall used, {remaining} remaining.
```

At ninety percent of the allowance, one further message requires the model to submit.

```
! FINISH NOW: budget is nearly exhausted. On this turn you must submit
your best answer by setting answer["content"] and answer["ready"] = True.
If you do not, the system will coerce a final answer, which may not be
your preferred one.
```

The only difference from the messages the other agents receive is that the coercion names the `answer` dict, since RLM submits that way rather than by calling `finish`.

M Implementation details: Claude Code and Codex

Both coding agents are run in headless mode, `claude -p` and `codex exec`, and access the same per-problem container instance the Meta-Reasoning Agent and Direct Control Agent workers use through a single MCP tool, `container_bash`. That tool reuses the same bash handler a worker gets, so both agents see identical output truncation, memory limits, and recovery behavior.

Their native tools are disabled so that every action flows through that one channel. Claude Code runs with its built-in tools off and only the MCP tool allowed; Codex runs with `-sandbox read-only`, so its own shell cannot write anything scored, with the MCP tool approved per-call. File edits therefore happen through heredocs and shell commands, exactly as they do for the Meta-Reasoning Agent and Direct Control Agent.

As with the other agents and systems, we keep these agents aware of the model-call budget. Each time another tenth of the allowance is consumed, a budget line is prefixed to the next tool result.

```
BUDGET: {used}/{budget} llmcall used, {remaining} remaining.
```

The system prompt also states the allowance and that every turn costs one call.