GATETREE: GATED CONTEXT SHARING FOR ADAPTIVE MULTI-AGENT COLLABORATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Parallel multi-agent systems share intermediate findings to coordinate different solution paths. As these paths evolve, however, shared information can become outdated or inapplicable, propagating unsupported assumptions and causing redundant work. Effective collaboration requires selecting information that remains relevant to each branch and adapting its plan accordingly. We introduce GateTree, a multi-agent framework that couples shared-context management with adaptive planning. GateTree maintains two concurrent branches with complementary conservative and exploratory strategies. A GateAgent checks the provenance and scope of shared records and selects context for each branch. An adaptive planner then uses this context and execution feedback to revise branch objectives and verification targets, preserving useful progress while addressing unresolved requirements. Across SWE-bench Pro, text-only HLE, and GAIA2, GateTree achieves the highest average Pass@2 under both backbone models while reducing cost by an average of 30.17% relative to each setting's strongest Pass@2 baseline. Replacing active context management with passive sharing reduces Pass@2 by 11.50 points in the component ablation. Further analysis shows that selective delivery is most beneficial for long contexts with sparse relevant evidence, whereas full-context delivery can be more effective when relevant evidence is dense. The code is provided in the supplementary materials.

1 INTRODUCTION

Parallel multi-agent systems pursue different solution paths for the same task and share intermediate findings to build on one another's progress (Mao & Mirhoseini, 2026; Hu et al., 2026). As agents revise their hypotheses and candidate solutions, however, the applicability of these findings changes. Evidence supporting one branch may be outdated or rest on assumptions rejected by another. Shared information therefore has value only in relation to the recipient's current state and intended actions. Effective coordination requires deciding which evidence remains applicable and how it should shape each branch's next decision.

Existing frameworks organize collaboration through roles, communication, and delegation. CAMEL and AutoGen support role-based cooperation and agent conversations (Li et al., 2023; Wu et al., 2024), while AOrchestra delegates subtasks to agents with tailored instructions, context, tools, and models (Ruan et al., 2026). Among concurrent systems, DeLM coordinates workers through verified shared progress and a task queue (Mao & Mirhoseini, 2026), and AgentFugue provides selective access to teammate reasoning episodes (Hu et al., 2026). These mechanisms make peer progress available for collaboration. A central challenge is maintaining the connection between that progress and evolving branch decisions: shared evidence should guide plan revision, while the revised plan should determine which evidence to deliver next. This feedback loop places evidence applicability and adaptive planning at the center of coordination.

Software repair makes this dependency concrete. A passing test supports the tested patch; its relevance must be reassessed when the patch changes or another branch develops a different candidate. Even applicable evidence must be translated into action. AgentFugue reports a case in which agents converge on an answer despite disqualifying evidence in the shared hub (Hu et al., 2026). Coordination must therefore preserve the scope of shared findings and carry their implications into execution and verification. Our trajectory analysis further exposes this challenge. Across two models and three

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

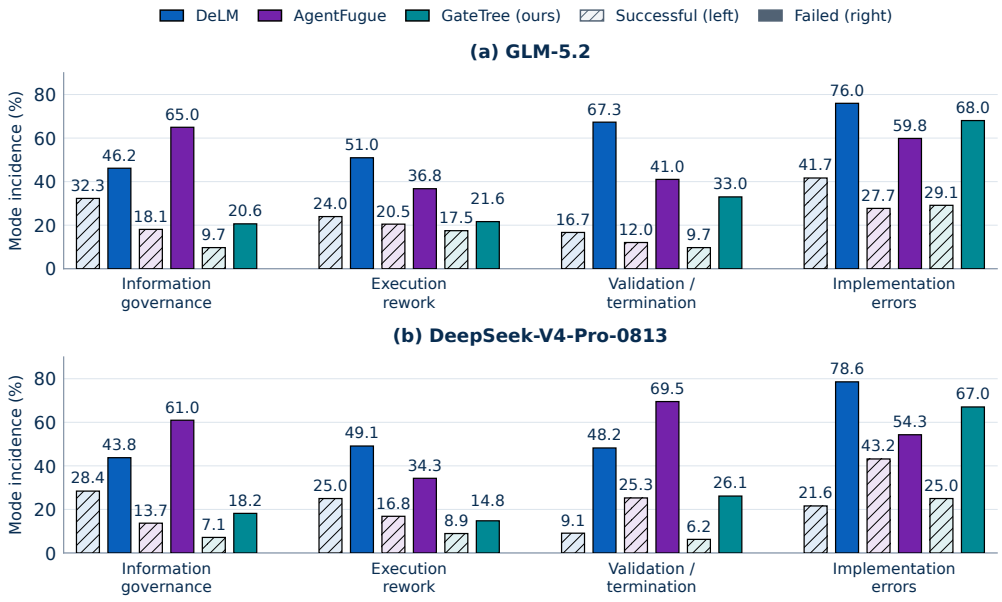

Figure 1: Failure-mode incidence on 200 SWE-bench Pro tasks with two backbone models. GateTree has the lowest incidence in all three coordination categories, including information governance, execution rework, and validation or termination, across both models and outcome groups. The fourth category captures implementation errors during model execution.

frameworks on the same 200 SWE-bench Pro tasks, we identify information-governance problems, execution rework, validation or termination problems, and implementation errors in both successful and failed trajectories (Figure 1). The first three categories concern framework coordination; the last concerns implementation during model execution. These findings motivate a framework that manages shared evidence and branch plans as interdependent parts of the execution process.

We introduce GateTree, a framework that integrates shared-context management with adaptive planning across two persistent branches (Figure 2). Its central mechanism is a feedback loop between evidence selection and plan revision. A task-local GateAgent checks record provenance and scope, maintains record states, and selects evidence for each recipient. The Planner combines this evidence with execution feedback to revise objectives, assumptions, and verification targets; the updated plan then guides subsequent evidence delivery. A conservative branch refines the best-supported candidate, while an exploratory branch investigates alternatives. This design turns shared progress and unresolved failures into concrete planning decisions, allowing branches to build on peer findings while preserving distinct candidate solutions and separate submissions.

Our contributions are threefold:

- We introduce GateTree, a multi-agent framework that couples evidence selection and plan revision in a feedback loop: shared evidence guides each branch's next objective and verification targets, while the revised plan determines which evidence to deliver next. Conservative and exploratory branches use this loop to reuse progress while preserving distinct solution paths.
- GateAgent makes shared-evidence applicability explicit and recipient-dependent. It separates record admission and lifecycle management from delivery-time eligibility, checking provenance and scope against each branch's current state. This separation allows a finding to remain available to branches for which it is valid while excluding it from those for which it no longer applies.
- Experiments on three benchmarks with two backbone models demonstrate the highest benchmark-averaged Pass@2 under both backbones, with an average cost reduction of 30.17% relative to each setting's strongest Pass@2 baseline. Factorial ablations show that managed context amplifies the benefit of adaptive planning, while controlled studies identify the conditions where selective delivery is most effective.

2 Related Work

Multi-agent collaboration and orchestration. Multi-agent systems coordinate specialized agents through structured communication and delegation. CAMEL and AutoGen establish role-based cooperation and configurable interaction patterns (Li et al., 2023; Wu et al., 2024), while MetaGPT and ChatDev organize software development into workflows with specialized responsibilities (Hong et al., 2024; Qian et al., 2024). THREAD and AOrchestra support more flexible delegation through recursive reasoning threads and task-specific sub-agents (Schroeder et al., 2025; Ruan et al., 2026). Most closely related to our work, DeLM coordinates concurrent agents through verified progress and dependency-aware task management (Mao & Mirhoseini, 2026), and AgentFugue enables selective access to teammate reasoning episodes (Hu et al., 2026). GateTree addresses the changing applicability of that progress: as concurrent branches develop different solutions, shared evidence must be checked against the recipient's current state and incorporated into its subsequent plan.

Memory and context management. Retrieval-augmented generation and LongMem extend model context through external passages and stored representations of past inputs (Lewis et al., 2020; Wang et al., 2023). MemGPT manages information across memory tiers (Packer et al., 2024); ReadAgent and Context-Folding reduce context demands through gist memories and summaries of completed work (Lee et al., 2024; Sun et al., 2025). Mem0 and A-Mem further support memory consolidation and updates (Chhikara et al., 2025; Xu et al., 2025). Yet retaining information does not guarantee its effective use, particularly in long contexts (Liu et al., 2024). In concurrent execution, relevance is also recipient-dependent: evidence that supports one branch may be outdated or inapplicable to another. GateTree therefore combines record lifecycle management with branch-specific eligibility checks and selective delivery, allowing shared information to track evolving execution states.

Planning with execution feedback. ReAct integrates reasoning with actions and observations (Yao et al., 2023), while Reflexion uses feedback-derived reflections to improve subsequent attempts (Shinn et al., 2023). SWE-agent and OpenHands support interactive execution through agent–computer interfaces and development environments (Yang et al., 2024; Wang et al., 2025). At the workflow level, AFlow uses execution feedback to search for effective agentic workflows (Zhang et al., 2025). GateTree focuses on online planning across concurrent solution paths. Its planner combines local execution outcomes with applicable peer evidence to revise branch objectives and verification targets. Revised plans in turn guide context delivery, so evidence selection and planning respond to one another throughout execution.

3 Method

3.1 Private branches and shared context

GateTree maintains two concurrent branches: a conservative branch refines the best-supported candidate, while an exploratory branch investigates alternatives. Each has its own worker, private history, plan, candidate, and submission. As these candidates evolve, a finding useful to one branch may no longer apply to another. A task-local, LLM-based GateAgent manages which shared records should inform each branch's next decision.

Let S_t^b and P_t^b denote the state and plan of branch $b \in \{A, B\}$ at coordination event t, and $\mathcal{M}_t$ the managed context. A plan specifies an objective, assumptions, actions, expected evidence, and verification requirements. Workers return local outcomes and proposed shared records, whose admission and delivery GateAgent controls.

Without initial shared evidence, the Planner dispatches directly to workers. After a branch returns an outcome, GateAgent consolidates available records into a review capsule for the Planner. The Planner revises the branch plan using this capsule and its local outcome; GateAgent then delivers the updated plan and a tailored execution capsule to the worker. Branches proceed asynchronously, with worker context refreshed as evidence arrives. This cycle couples two decisions: which shared findings still apply, and how the branch should proceed given those findings. Evidence sources and completion checks vary by task; Appendix E describes their instantiation for software repair and tool-free reasoning.

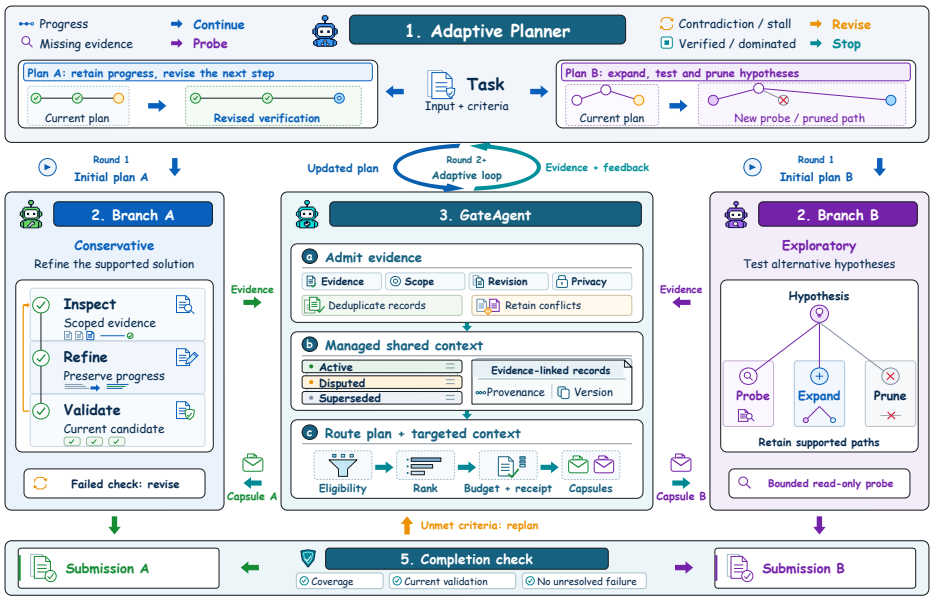

Figure 2: Overview of GateTree. An adaptive Planner coordinates two concurrent branches with complementary conservative and exploratory strategies. Initial plans are dispatched directly to the workers. In subsequent coordination cycles, GateAgent checks the provenance and applicability of shared records, maintains their lifecycle states, and delivers recipient-specific context. Review capsules and local execution feedback guide plan revision, while updated plans inform context selection for continued worker execution. Each branch maintains its own candidate solution and submits separately after completion checks.

3.2 Record Lifecycle and Applicability

A record contains its type, content, evidence, scope, branch provenance, version, status, and repository revision when applicable. Types distinguish facts, observations, attempted approaches, failures, claims, and candidate progress. These fields tie evidence to its conditions of use: a test result refers to the candidate at revision r, and a derived conclusion depends on its stated premises. Reuse requires checking those conditions against the recipient's current state.

Deterministic admission checks validate format, require evidence references for designated record types, reject recognized private-evaluation records and obsolete versions, and remove exact duplicates. GateAgent then reconciles records within their declared scope. Conflicting claims are marked disputed, while an accepted replacement supersedes its predecessor. Both remain in the record history but are excluded from ordinary delivery. This distinction separates unresolved disagreement from an update that replaces earlier information. Before delivery, active records must satisfy type, role, phase, and cross-branch safety constraints. Supporting evidence must match the record's declared scope, and shared claims retain their provenance. A change to the recipient's candidate can make a record inapplicable without superseding it for other recipients.

3.3 Recipient-Specific Delivery

For a recipient with role a, phase p, branch b, and revision r, GateAgent constructs an eligible record set $\mathcal{E}_t(a, p, b, r)$. Delivery receipts suppress previously delivered records, with a limited repeat allowance for high-signal evidence. Delivery history and record validity are tracked separately.

GateAgent ranks eligible records and packs them into a context capsule:

$$K_t^{a,b} = \mathrm{Pack}_{W_t^{a,b}}\big(\mathrm{Rank}(\mathcal{E}_t(a, p, b, r))\big), \qquad \widehat{\mathrm{tok}}(K_t^{a,b}) \le W_t^{a,b},$$

where $W_t^{a,b}$ is the delivery budget and $\widehat{\mathrm{tok}}$ is the configured token counter. This budget covers delivered record text; the worker's full prompt and private history are accounted for separately.

Selection reflects the current task, recipient role, execution phase, and branch plan. The Planner receives evidence for reviewing progress and unresolved requirements, while the worker receives evidence relevant to executing the updated plan. GateAgent uses LLM-based selection to choose records that support these decisions within the delivery budget. Lifecycle management tracks which records remain active; plan-conditioned selection turns this shared state into evidence for the next branch decision. The selection interface and fallback rules are specified in Appendix F.

3.4 Feedback-driven branch planning

After the initial dispatch, the Planner updates each branch plan using a fresh review capsule and the latest local outcome z_t^b:

$$P_{t+1}^b = \mathrm{Update}\big(P_t^b, K_t^{\mathrm{planner},b}, z_t^b\big).$$

The outcome summarizes execution status, new evidence, validation results, and unresolved requirements. The review capsule supplies applicable peer findings alongside evidence of remaining problems. Together, they allow the Planner to retain supported progress while redirecting work toward an unresolved assumption or check. The revised plan then determines which records GateAgent selects for the worker, closing the loop between shared evidence and branch execution.

A branch-local regulator checks critical assumptions before execution. Unsupported premises require revision, further derivation, or a bounded read-only probe when tools are available. Separate plan histories prevent a rejected hypothesis from automatically invalidating the peer's plan. After an incomplete result, the Planner revises objectives, assumptions, actions, or verification targets while retaining supported progress. Conservative execution refines the candidate or its checks; exploratory execution expands promising alternatives and stops contradicted or dominated directions. These decisions use recorded progress, candidate validity, and remaining requirements to guide continued execution.

Submission checks deliverable validity, requirement coverage, and outstanding verification obligations using the feedback available in the task. Branches submit their own candidates after the task-specific completion procedure; the held-out evaluator determines final success. Algorithm 1 summarizes the cycle, with implementation details in Appendix E.

4 Experiments

4.1 Evaluation settings

Benchmarks. We evaluate GateTree on three benchmarks spanning software engineering, expert-level reasoning, and dynamic agent interaction: SWE-bench Pro's 731 public repository-level issue-resolution tasks (Deng et al., 2025), HLE's 2,158 text-only questions evaluated without tools (Phan et al., 2025), and GAIA2's 800 scenarios, with 160 scenarios each for ambiguity, adaptability, execution, search, and time (Froger et al., 2026).

Baselines. We compare five baselines. ReAct interleaves reasoning and action (Yao et al., 2023), OpenClaw (2026.8.1) provides a general-purpose agent framework (OpenClaw contributors, 2026), and mini-SWE-agent v2 (2026.8.1) is a lightweight software-engineering scaffold (Yang et al., 2024). DeLM and AgentFugue are the closest shared-context baselines. DeLM combines verified progress with task-queue coordination (Mao & Mirhoseini, 2026). AgentFugue supports selective access to peer reasoning through a shared hub (Hu et al., 2026), and we replace its original trained hub model with the evaluation backbone while retaining the remaining framework settings.

Models and implementation. We use GLM-5.2 and DeepSeek-V4-Pro-0813 (DeepSeek), deployed on eight NVIDIA H20 GPUs. All frameworks use temperature 1.0, top-p 1.0, a 32K-token generation limit per call, a 400K-token context window, and maximum thinking effort. For the main comparison, we report means and standard deviations over three repeated evaluations, each with two submissions per task. Pass@2 measures success in at least one submission, and Avg. measures the mean success rate across the two submissions. Cost is mean inference cost per task, with token usage converted to USD using the fixed rates in Appendix E.5.

Framework budgets. DeLM's original SWE-bench Verified configuration allows up to 50 agent–environment steps and six internal attempts (Mao & Mirhoseini, 2026). To accommodate the more demanding tasks in SWE-bench Pro, we increase these limits to 75 steps and eight internal attempts and apply the same limits across the compared frameworks. Each framework produces two submitted attempts per task. ReAct, OpenClaw, and mini-SWE-agent generate independent trajectories; DeLM runs two concurrent solvers, matching GateTree's two execution branches. AgentFugue likewise produces two submissions, with internal peer teams that vary by benchmark. This setup aligns submission and execution budgets while preserving each framework's coordination mechanism. We also analyze branch scalability and role complementarity in Appendix C, comparing GateTree with two, four, and eight parallel branches against one to eight independent attempts.

4.2 Main results

Table 1: Performance and inference cost across three benchmarks and two backbone models. Results are reported as mean ± standard deviation over three repeated evaluations. Pass@2 and Avg. are percentages; Cost is the mean inference cost per task in USD. Overall reports equally weighted means across the benchmarks. Bold and underline indicate the best and second-best means, respectively.

	SWE-bench Pro			HLE text-only			GAIA2			Overall		
Method	$P@2$	Avg.	Cost	$P@2$	Avg.	Cost	$P@2$	Avg.	Cost	$P@2$	Avg.	Cost
GLM-5.2												
ReAct	30.46 ± 1.14	22.09 ± 0.87	<u>1.18</u> ± 0.06	39.36 ± 0.94	32.87 ± 0.79	**0.34** ± 0.02	41.25 ± 1.19	35.60 ± 1.03	<u>0.78</u> ± 0.04	37.02	30.19	**0.77**
OpenClaw	31.87 ± 1.23	27.93 ± 1.02	1.76 ± 0.09	43.57 ± 1.03	36.74 ± 0.86	0.47 ± 0.03	43.83 ± 1.26	39.69 ± 1.09	0.83 ± 0.05	39.76	34.79	1.02
mini-SWE-agent	<u>57.73</u> ± 0.96	**50.50** ± 0.91	2.37 ± 0.13	—	—	—	—	—	—	—	—	—
DeLM	53.67 ± 1.16	49.86 ± 1.04	1.81 ± 0.08	49.63 ± 1.08	**44.72** ± 0.93	<u>0.35</u> ± 0.02	51.67 ± 1.32	40.96 ± 1.14	0.81 ± 0.04	51.66	45.18	0.99
AgentFugue	50.75 ± 1.30	45.62 ± 1.10	**1.07** ± 0.05	<u>51.48</u> ± 0.97	<u>44.61</u> ± 0.89	0.71 ± 0.04	<u>55.79</u> ± 1.16	**45.71** ± 1.02	0.79 ± 0.04	<u>52.67</u>	<u>45.31</u>	<u>0.86</u>
GateTree (ours)	**59.19** ± 0.93	<u>50.21</u> ± 0.94	1.19 ± 0.05	**57.09** ± 0.83	44.20 ± 0.87	0.38 ± 0.02	**57.13** ± 0.98	<u>44.00</u> ± 1.06	**0.73** ± 0.03	**57.80**	**46.14**	**0.77**
DeepSeek-V4-Pro-0813												
ReAct	31.87 ± 1.12	24.17 ± 0.93	**0.42** ± 0.02	47.07 ± 0.92	37.98 ± 0.81	**0.08** ± 0.01	40.17 ± 1.21	29.98 ± 1.03	0.45 ± 0.03	39.70	30.71	<u>0.32</u>
OpenClaw	39.95 ± 1.19	31.24 ± 1.06	0.82 ± 0.04	48.18 ± 1.02	42.73 ± 0.87	0.15 ± 0.01	40.92 ± 1.23	30.85 ± 1.07	0.54 ± 0.03	43.02	34.94	0.50
mini-SWE-agent	**61.24** ± 0.97	**53.51** ± 0.94	0.76 ± 0.04	—	—	—	—	—	—	—	—	—
DeLM	54.08 ± 1.14	44.51 ± 1.01	0.65 ± 0.03	55.64 ± 1.04	52.80 ± 0.92	0.18 ± 0.01	<u>47.21</u> ± 1.18	**43.21** ± 1.01	0.49 ± 0.03	<u>52.31</u>	**46.84**	0.44
AgentFugue	54.40 ± 1.23	43.91 ± 1.07	0.48 ± 0.03	<u>56.53</u> ± 0.99	<u>52.91</u> ± 0.86	0.12 ± 0.01	43.17 ± 1.28	<u>40.75</u> ± 1.09	**0.39** ± 0.02	51.37	45.86	0.33
GateTree (ours)	<u>60.28</u> ± 0.89	<u>46.01</u> ± 0.98	<u>0.44</u> ± 0.02	**60.09** ± 0.84	**53.10** ± 0.91	<u>0.10</u> ± 0.01	**51.63** ± 0.98	40.60 ± 1.03	<u>0.40</u> ± 0.02	**57.33**	<u>46.57</u>	**0.31**

Table 1 shows that GateTree combines broad task coverage with efficient inference across software engineering, reasoning, and dynamic interaction. GateTree leads Overall Pass@2 under both backbones, with lower cost than the strongest Pass@2 baseline in each setting and an average reduction of 30.17%. It ranks first in Overall Avg. with GLM-5.2 and second with DeepSeek, reaching 46.14% and 46.57%, respectively. On SWE-bench Pro, GateTree exceeds mini-SWE-agent by 1.46 Pass@2 points with GLM-5.2 at 49.79% lower cost. With DeepSeek, it achieves 60.28% compared with mini-SWE-agent's 61.24%, while reducing cost by 42.11%. Its shared coordination framework also supports tool-free reasoning and dynamic interaction, extending beyond the software-engineering evaluation of mini-SWE-agent. GateTree also exceeds both shared-context baselines, DeLM and AgentFugue, in Pass@2 and ranks second in Avg. under both backbones. These results show strong repair coverage at substantially lower inference cost than the leading software-engineering scaffold.

The coverage gains are larger on text-only HLE, where GateTree improves over AgentFugue by 5.61 and 3.56 percentage points while lowering cost. With DeepSeek, it also achieves the highest Avg. at 53.10%. GateAgent manages evolving derivations and critiques, allowing branches to revise their reasoning using applicable peer evidence. These gains demonstrate effective shared-context coordination in a tool-free setting, extending the framework beyond executable feedback. On GAIA2, GateTree leads Pass@2 under both backbones at lower cost than the strongest Pass@2 baseline, extending its coverage–cost advantage to dynamic interaction. The following ablations examine how context management and adaptive planning contribute to these gains, both individually and through their interaction.

5 ANALYSIS

5.1 COMPONENT ABLATION

We evaluate component variants on 200 SWE-bench Pro tasks using DeepSeek. Full combines differentiated branches, managed shared context, and adaptive planning, achieving 54.50% Pass@2 at a mean cost of $0.40 per task. Passive Shared Context replaces GateAgent with an append-only chronological board, jointly removing admission checks, recipient-specific routing, deduplication, and priority selection. No Shared Context disables cross-branch delivery; Homogeneous-2 gives both workers the same profile; and Static Planner executes the initial plans without replanning. Combining passive sharing with static planning completes a 2×2 comparison.

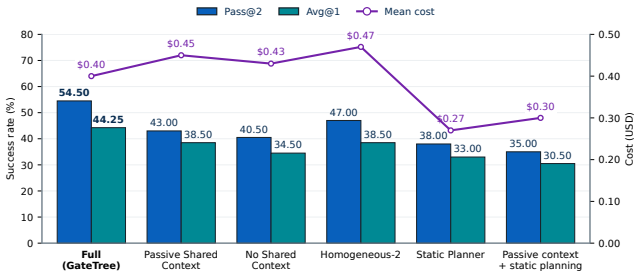

Figure 3: Component ablation on 200 SWE-bench Pro tasks with DeepSeek. Bars show Pass@2 and average submission success (Avg@1); the line shows mean cost.

Figure 3 shows that each component contributes to performance. Passive sharing improves over independent branches; managed context adds another 11.50 Pass@2 points and reduces mean cost from $0.45 to $0.40. This gain reflects the combined contribution of admission, selection, and delivery to effective reuse of shared progress. Differentiated branches outperform homogeneous workers by 7.50 points, supporting the combination of conservative and exploratory strategies. Removing replanning causes the largest single-component decline, alongside lower computation cost. The factorial comparison further shows that adaptive planning improves Pass@2 by 16.50 points with GateAgent, compared with 8.00 points under passive sharing. Managed context therefore increases the benefit of replanning by 8.50 points, demonstrating the value of coupling evidence management with branch decisions. Appendix B presents more comprehensive component ablations.

5.2 CONTEXT MANAGEMENT EFFECTIVENESS

Effective sharing requires selecting evidence that supports the recipient's next decision. We examine this capability through two studies: a comparison of context-selection policies and a controlled analysis of source length and evidence density. Both follow the component ablation's task subset, backbone, and evaluation settings.

Dynamic context selection. To assess the value of recipient-aware selection, we compare GateAgent with three alternatives. Recent selects the newest records, while Priority uses a fixed ordering of patch progress, failures, observations, and facts. GateAgent adapts selection to the current task, recipient role, and execution phase. All three use a 2K-token delivery budget, excluding the worker's private context. All Context instead delivers every record that passes safety, branch, and code-version checks, without a delivery limit. This design tests whether dynamic selection offers benefits beyond restricting context length or exposing all eligible information.

Table 2: GateAgent improves success and efficiency over alternative context-selection policies.

Method	Budget	Pass@2	Attempts	Steps	Cost
Recent	2K	44.50%	6.32	52.32	0.48
Priority	2K	49.50%	6.11	51.66	0.45
GateAgent	2K	**54.50%**	**5.88**	**48.84**	**0.40**
All Context	Unlimited	49.00%	6.70	56.46	0.67

GateAgent achieves the highest Pass@2 and the lowest mean attempts, steps, and cost (Table 2). Under the same 2K budget, it outperforms Recent and Priority by 10.00 and 5.00 percentage points, respectively, demonstrating the value of adapting delivery to the recipient beyond limiting context length. It also exceeds All Context by 5.50 points while reducing mean cost from $0.67 to $0.40 and requiring fewer attempts and steps. Selecting

among eligible records therefore improves both success and efficiency over delivering the entire eligible history.

Robustness to distracting context. We next vary the amount and composition of available information to identify when selective delivery is most effective. Direct Read receives the complete source context each round, whereas GateAgent selects a 2K–16K-token capsule for the worker. Relevant content consists of target-task observations collected before the first source edit, including file reads, searches, and call-relationship outputs. Distractors contain the same observation types from other tasks, preferentially from the same repository, and can therefore share symbols, structures, and terminology with the target. Relevance is defined by task provenance: target-task observations constitute relevant content, and observations from other tasks constitute distractors. Using the DeepSeek tokenizer, we construct twelve conditions with source lengths $L \in \{8\text{K}, 32\text{K}, 96\text{K}\}$ and relevant-content densities $D \in \{5\%, 10\%, 20\%, 50\%\}$. For example, the 32K, 20% condition contains 32,768 tokens: 6,554 from the target task and 26,214 from other tasks. The two methods receive identical sources for each task and condition; Appendix A details the construction.

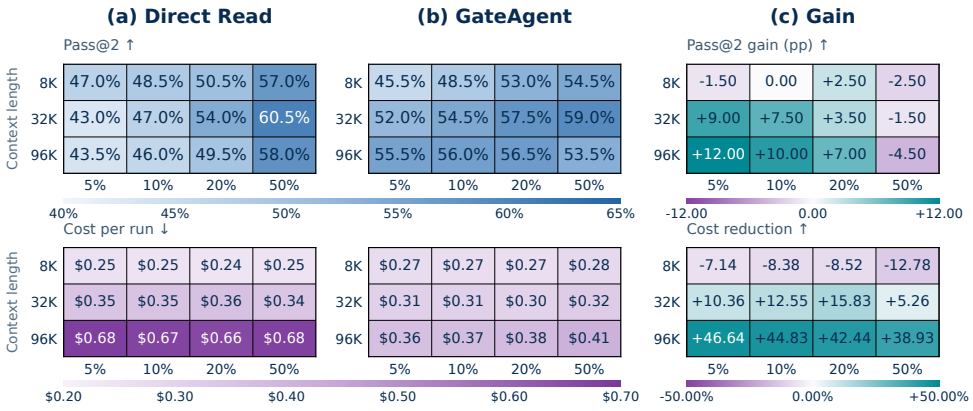

Figure 4: Selective delivery improves the success–cost trade-off most in long contexts with sparse relevant evidence. Gain reports the Pass@2 difference in percentage points and relative cost reduction against Direct Read; positive values favor GateAgent.

GateAgent's advantage grows as the source context becomes longer and relevant evidence more dispersed (Figure 4). At 32K, it improves macro-average Pass@2 by 4.63 points while reducing cost by 11.07%; at 96K, these gains reach 6.13 points and 43.22%, respectively. The largest accuracy improvement occurs at 96K and 5% density, where GateAgent achieves 55.50% Pass@2 compared with 43.50% for Direct Read. Across densities of 5%–20%, selective delivery becomes increasingly effective as shared history grows, concentrating worker context on evidence from the target task.

Source density determines the trade-off between selective and full-context delivery. GateAgent provides its strongest success–cost gains for long, sparse histories. Direct Read is more accurate at 50% density across all lengths, while GateAgent remains cheaper at 32K and 96K; at 8K, selection adds cost without an aggregate accuracy gain. These results identify when selection is most useful and when preserving more of the available evidence improves accuracy.

5.3 Failure modes and trajectory analysis

Figure 1 compares four non-exclusive problem categories: information governance, execution rework, validation or termination, and implementation errors. We analyze 1,200 two-branch trajectories covering the same 200 SWE-bench Pro tasks under two models and three frameworks. Successful trajectories may contain problems resolved later or confined to one branch. The analysis protocol and category definitions are provided in Appendix D. GateTree has the lowest incidence in all three coordination categories across both models and outcome groups. Among failed DeepSeek trajectories, information-governance incidence is 18.2%, compared with 43.8% for DeLM and 61.0% for AgentFugue; validation or termination rates are 26.1%, 48.2%, and 69.5%, respectively. GLM shows the same pattern. Failed trajectories exhibit more problems in every category, with

larger success–failure gaps in implementation errors for DeLM and in validation or termination for AgentFugue. These results highlight GateTree's strengths in managing shared information and coordinating execution across both models.

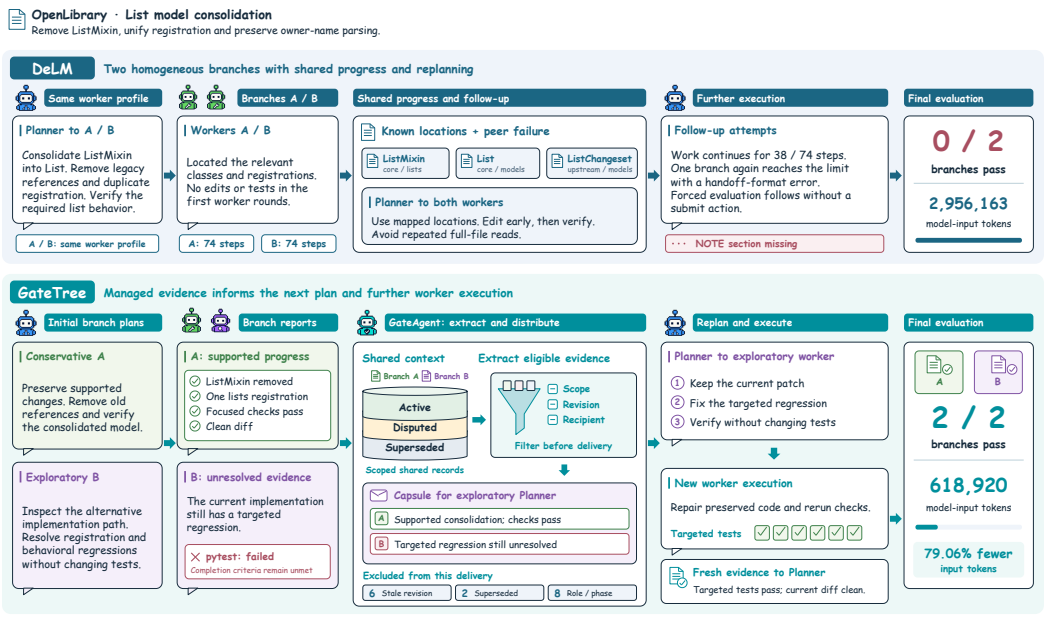

Figure 5: Managed evidence guides targeted replanning in a SWE-bench Pro case with DeepSeek. DeLM fails in both branches, whereas GateTree passes in both with fewer input tokens. Dialogue paraphrases selected events; token counts cover cumulative model inputs over each full run.

Figure 5 compares coordination trajectories on a SWE-bench Pro OpenLibrary task involving list-model consolidation. DeLM's two homogeneous workers initially locate relevant code but complete neither edits nor tests. The Planner forwards these locations and peer failures, instructing workers to edit early and avoid repeated full-file reads. Execution nevertheless remains incomplete: one branch reaches the step limit with a format error, and forced evaluation proceeds without an explicit submission. Neither candidate passes. Shared findings accumulate without a verified repair.

GateTree uses shared evidence to guide targeted replanning. The conservative branch reports a consolidation supported by focused checks, while the exploratory branch identifies a remaining regression. GateAgent combines the peer's scoped patch summary with local failure evidence, excluding stale, superseded, and recipient-ineligible records. The exploratory branch's revised plan preserves its candidate, targets the regression, and specifies checks for the affected behavior. Subsequent execution passes these checks, and both branches pass final evaluation with 79.06% fewer input tokens than DeLM. This trajectory shows how scoped evidence and adaptive planning preserve useful progress and direct execution toward unresolved requirements.

6 Conclusion

GateTree integrates shared-context management with adaptive planning through a feedback loop between evidence selection and plan revision. GateAgent selects applicable evidence, and the Planner combines it with execution feedback to revise objectives and verification targets. Updated plans guide subsequent delivery, allowing complementary branches to reuse progress while preserving distinct solution paths. Across three benchmarks, GateTree achieves the highest benchmark-averaged Pass@2 under both backbones, with 30.17% lower cost on average than the strongest Pass@2 baseline in each setting. Ablations show that managed context strengthens replanning, while controlled studies demonstrate the benefits of selective delivery for long, sparse histories. These findings highlight shared-context management and selective information delivery as key mechanisms for effective multi-agent collaboration.

AI USE STATEMENT

Generative AI tools assisted language polishing and figure preparation, including improvements to clarity, grammar, and visual layout. The authors reviewed and revised the resulting text and figures.

REPRODUCIBILITY STATEMENT

We provide our implementation in the supplementary materials to support reproducibility. Section 4 specifies the benchmarks, model configurations, execution budgets, and evaluation protocol. GateTree's coordination procedure and implementation details are described in Algorithm 1 and Appendix E, with role prompts and response formats in Appendix F. The appendices also document the construction of controlled contexts, ablation configurations, and trajectory-analysis protocol used in our studies.

REFERENCES

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory, 2025. URL https://arxiv.org/abs/2504.19413.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks? *arXiv preprint arXiv:2509.16941*, 2025. doi: 10.48550/arXiv.2509.16941. URL https://arxiv.org/abs/2509.16941.

Romain Froger, Pierre Andrews, Matteo Bettini, Amar Budhiraja, Ricardo Silveira Cabral, Virginie Do, Emilien Garreau, Jean-Baptiste Gaya, Hugo Laurençon, Maxime Lecanu, Kunal Malkan, Dheeraj Mekala, Pierre Ménard, Gerard Moreno-Torres Bertran, Ulyana Piterbarg, Mikhail Plekhanov, Mathieu Rita, Andrey Rusakov, Vladislav Vorotilov, Mengjue Wang, Ian Yu, Amine Benhalloum, Grégoire Mialon, and Thomas Scialom. Gaia2: Benchmarking LLM Agents on Dynamic and Asynchronous Environments. In *International Conference on Learning Representations*, 2026. doi: 10.48550/arXiv.2602.11964. URL https://arxiv.org/abs/2602.11964.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework, 2024. URL https://arxiv.org/abs/2308.00352.

Yuyang Hu, Hongjin Qian, Shuting Wang, Jiongnan Liu, Tong Zhao, Xiaoxi Li, Zheng Liu, and Zhicheng Dou. AgentFugue: Agent Scaling for Long-Horizon Tasks through Collective Reasoning. *arXiv preprint arXiv:2605.24486*, 2026. doi: 10.48550/arXiv.2605.24486. URL https://arxiv.org/abs/2605.24486.

Kuang-Huei Lee, Xinyun Chen, Hiroki Furuta, John Canny, and Ian Fischer. A Human-Inspired Reading Agent with Gist Memory of Very Long Contexts, 2024. URL https://arxiv.org/abs/2402.09727.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems*, volume 33, pp. 9459–9474. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society, 2023. URL https://arxiv.org/abs/2303.17760.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. doi: 10.1162/tacl_a_00638. URL https://aclanthology.org/2024.tacl-1.9/.

Yuzhen Mao and Azalia Mirhoseini. Decentralized Multi-Agent Systems with Shared Context. *arXiv preprint arXiv:2606.10662*, 2026. doi: 10.48550/arXiv.2606.10662. URL https://arxiv.org/abs/2606.10662.

OpenClaw contributors. OpenClaw. https://github.com/openclaw/openclaw, 2026. Software repository, accessed 2026-09-07.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as Operating Systems, 2024. URL https://arxiv.org/abs/2310.08560v2.

Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Chen Bo Calvin Zhang, Mohamed Shaaban, John Ling, Sean Shi, et al. Humanity's Last Exam. *arXiv preprint arXiv:2501.14249*, 2025. doi: 10.48550/arXiv.2501.14249. URL https://arxiv.org/abs/2501.14249.

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

Jianhao Ruan, Zhihao Xu, Yiran Peng, Fashen Ren, Zhaoyang Yu, Xinbing Liang, Jinyu Xiang, Yongru Chen, Bang Liu, Chenglin Wu, Yuyu Luo, and Jiayi Zhang. Aorchestra: Automating sub-agent creation for agentic orchestration. *arXiv preprint arXiv:2602.03786*, 2026. doi: 10.48550/arXiv.2602.03786. URL https://arxiv.org/abs/2602.03786.

Philip Schroeder, Nathaniel W. Morgan, Hongyin Luo, and James R. Glass. THREAD: Thinking Deeper with Recursive Spawning. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 8418–8442, Albuquerque, New Mexico, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.naacl-long.427. URL https://aclanthology.org/2025.naacl-long.427/.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling Long-Horizon LLM Agent via Context-Folding, 2025. URL https://arxiv.org/abs/2510.11967.

Weizhi Wang, Li Dong, Hao Cheng, Xiaodong Liu, Xifeng Yan, Jianfeng Gao, and Furu Wei. Augmenting Language Models with Long-Term Memory, 2023. URL https://arxiv.org/abs/2306.07174.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversations. In *Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BAakY1hNKS.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-Mem: Agentic Memory for LLM Agents. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 17577–17604. Curran Associates, Inc., 2025. doi: 10.52202/085713-0593. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/19909c36f51abc4856b4560aff3d36d6-Paper-Conference.pdf.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652, 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations*, 2023. doi: 10.48550/arXiv.2210.03629. URL https://arxiv.org/abs/2210.03629.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating Agentic Workflow Generation. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=z5uVAKwmjf.

APPENDIX CONTENTS

A Context Construction and Paired Comparisons **13**

B Extended Component Ablations **14**

B.1 Context management and adaptive planning 14

B.2 Applicability filtering and plan-conditioned selection 15

C Branch Scalability and Role Complementarity **16**

D Failure-Mode Analysis Protocol and Illustrative Cases **17**

E Implementation Contract and Scope **18**

E.1 Record admission and lifecycle 19

E.2 Recipient-specific selection and delivery 20

E.3 From shared evidence to revised branch plans 21

E.4 Shared evidence without executable feedback: HLE 22

E.5 Token-to-cost conversion 22

F Role Prompts and Response Formats **23**

F.1 Planner: evidence-conditioned decisions 23

F.2 Workers: differentiated search, common completion criteria 24

F.3 GateAgent: context selection 25

A CONTEXT CONSTRUCTION AND PAIRED COMPARISONS

The context study uses a fixed subset of 200 SWE-bench Pro tasks with DeepSeek and the main experiment's inference and execution settings. Target-task material consists of repository observations collected before the first source-code edit: file reads, searches, and call-relation outputs. Distractors use the same observation types from other tasks, with preference for tasks in the same repository. Shared symbols and terminology can therefore appear in both pools. Density measures the proportion of target-task material by provenance; the usefulness of individual observations can vary within both pools. Post-edit repair results are excluded from the target pool. Figure 6 shows how the pools are combined and delivered.

For source length L and target-task density D, we allocate tokens between the two pools as follows:

$$N_{\text{target}} = \text{round}(LD), \qquad N_{\text{other}} = L - N_{\text{target}}.$$

Tokens are counted with the DeepSeek tokenizer, with $\text{K} = 1024$. Thus, a 32K source at 20% density allocates 6,554 tokens to target-task observations and 26,214 to distractors. The twelve conditions combine 8K, 32K, and 96K source lengths with target-task densities of 5%, 10%, 20%, and 50%, reusing the same 200-task pool throughout. The allocation rule sets the target count in each condition; all remaining tokens come from other-task material.

Both context studies hold the task pool, backbone, and execution budget fixed while varying how shared material is selected or delivered. Within each robustness-study pair, Direct Read and GateAgent receive the same constructed source, with length and density fixed within each pair. In the selection-policy study, Recent, Priority, and GateAgent share the same 2K delivery window. All Context removes this additional cap while retaining eligibility checks. The shared-record delivery window is accounted for separately from the model context window and total inference use.

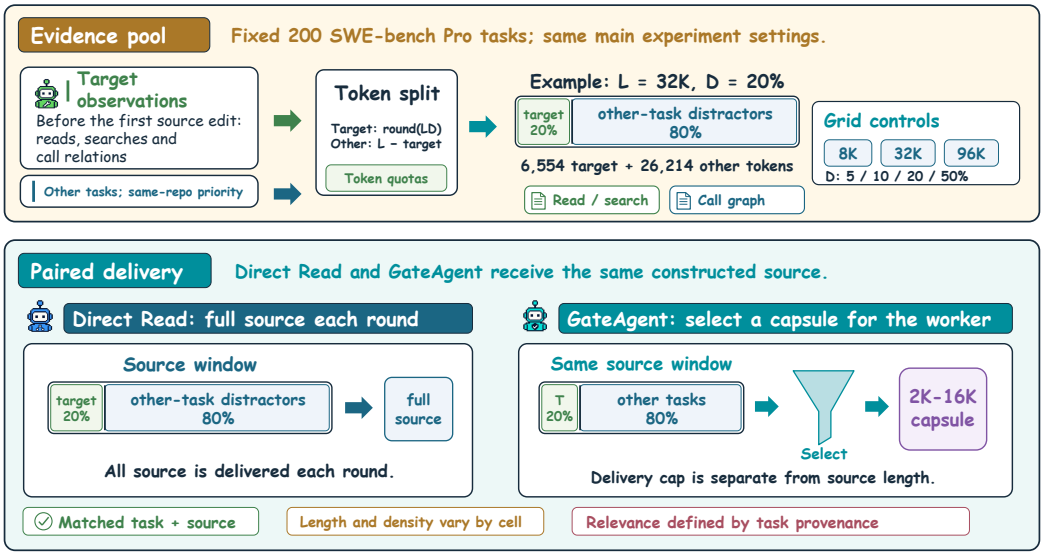

Figure 6: Construction of paired context conditions. Target-task observations before the first source edit are combined with observations from other tasks, preferentially in the same repository. The token counts illustrate the 32K, 20% condition. Direct Read and GateAgent receive the same constructed source; they differ in full-context delivery versus a bounded, selected capsule for the Worker.

B Extended Component Ablations

B.1 Context Management and Adaptive Planning

To examine the interaction between context management and adaptive planning, we extend the component ablation to a 2×2 factorial comparison by adding passive shared context with static planning to the three corresponding configurations in Figure 3. The two factors are context management (an append-only shared board or GateAgent) and planning adaptivity (fixed initial plans or plans revised using execution feedback). All configurations retain the conservative and exploratory branches and use the same 200 SWE-bench Pro tasks with DeepSeek, under the inference settings and resource limits of the component ablation.

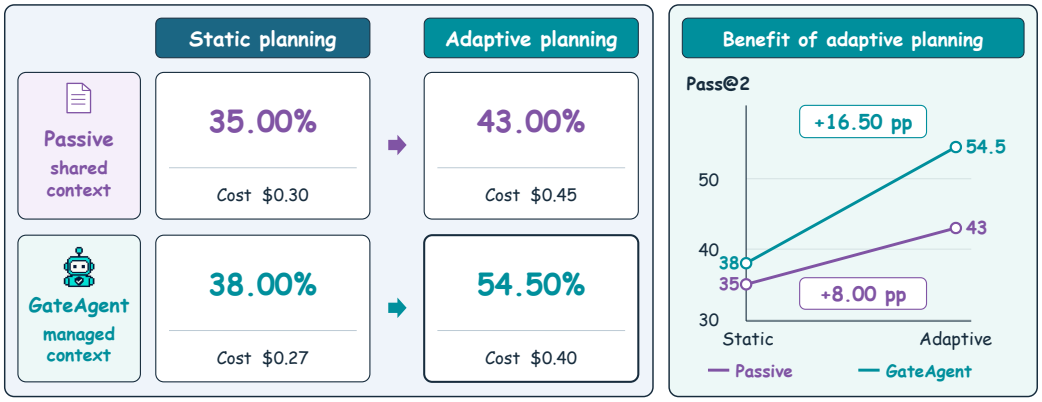

Figure 7: Interaction between context management and adaptive planning. Left: Pass@2 and mean cost per run for the four configurations. Right: the improvement from static to adaptive planning under each context policy.

We measure the interaction as the difference between the adaptive-planning gain under managed context and the same gain under passive shared context:

$$\Delta_{\text{coupling}} = \big(G_{\text{adaptive}} - G_{\text{static}}\big) - \big(P_{\text{adaptive}} - P_{\text{static}}\big),$$

where G and P denote task-level Pass@2 under GateAgent and passive sharing, respectively. A positive value means that adaptive planning provides a larger improvement when paired with managed context.

Figure 7 shows that adaptive planning improves Pass@2 under both context policies, with a larger improvement under GateAgent. Under passive sharing, Pass@2 increases from 35.00% to 43.00%; with GateAgent, it increases from 38.00% to 54.50%. The resulting interaction is 8.50 percentage points: adaptive planning gains more from managed evidence than from passive access to shared history. This interaction captures the additional benefit of coupling context management with plan revision.

B.2 Applicability filtering and plan-conditioned selection

We further separate two mechanisms within GateAgent: explicit applicability checks before delivery and selection conditioned on the updated branch plan. The study uses the same fixed 200 SWE-bench Pro task IDs and DeepSeek-V4-Pro-0813 backbone as the component ablation, with one conservative and one exploratory branch. Full and the three targeted variants retain the same adaptive Planner, including its model, prompts, and update rules. Assumption checks, current-candidate verification, requirement coverage, and submission conditions are also held fixed. Execution follows the same 75-step and eight-internal-attempt limits and counting conventions. Worker capsules use Table 2's 2K delivery budget, with unchanged token accounting, truncation, and handoff priority. Admission, record lifecycle management, deduplication, delivery receipts, retries, and fallback behavior are retained across these four configurations.

Table 3: Extended component ablation on 200 SWE-bench Pro tasks with DeepSeek. Pass@2 and Avg. are percentages; cost is mean complete inference cost per task in USD, including Planner and GateAgent calls where present. Bold marks the best value in each column.

Configuration	Pass@2 ↑	Avg. ↑	Cost ↓
Full GateTree	**54.50**	**44.25**	0.40
w/o Applicability Filter	49.50	40.75	0.44
w/o Plan-conditioned Selection	51.00	42.00	0.42
w/o Both	47.00	39.50	0.46
Passive Shared Context	43.00	38.50	0.45
No Shared Context	40.50	34.50	0.43
Homogeneous-2	47.00	38.50	0.47
Static Planner	38.00	33.00	**0.27**
Passive + Static	35.00	30.50	0.30

In w/o Applicability Filter, we remove delivery-time checks for recipient role and phase, branch and artifact-revision compatibility, and revision portability for cross-branch evidence. Records retain their provenance, revision, and scope metadata, which remain available to the semantic selector and Worker. Task boundaries, privacy and sharing authorization, private-evaluation exclusion, evidence-reference requirements, active-status checks, version management, and deduplication remain in place. This variant isolates the contribution of explicit applicability filtering alongside semantic selection and final verification of the current candidate.

In w/o Plan-conditioned Selection, the Planner still receives its review capsule, revises the plan, and sends that updated plan to the Worker. The intervention changes only the inputs used to select evidence for the Worker execution capsule. Writing x for the original task, q for the recipient information, and $\mathcal{E}$ for eligible records, the two selection rules are

$$K_{\text{full}} = \text{Pack}_{2\text{K}}\big(\text{Select}(\mathcal{E}, x, q, P^{b}_{t+1})\big),$$
$$K_{\text{no-plan}} = \text{Pack}_{2\text{K}}\big(\text{Select}(\mathcal{E}, x, q)\big).$$

The ablated selector receives the original task, recipient information, and candidate evidence, with plan-derived objectives, assumptions, actions, verification targets, and delegated task text omitted from its input. The same exclusion applies to the TASK field, delegated-objective fields, and candidate handoff text, preventing these inputs from reintroducing the updated plan. No initial or earlier plan is substituted. The current plan is still delivered to the Worker through the normal handoff path, with unchanged priority and budget accounting; Planner review is unchanged. This intervention tests whether evidence selection benefits from knowing what the Worker will do next while preserving adaptive planning itself.

The w/o Both configuration applies both interventions. It retains admission, record lifecycle management, deduplication, and semantic selection, isolating the combined contribution of applicability filtering and plan conditioning within the managed-context framework. The remaining rows in Table 3 reproduce the broader component variants for comparison.

Removing applicability filtering lowers Pass@2 by 5.00 percentage points and Avg. by 3.50 points, while increasing mean cost from \$0.40 to \$0.44. Explicit scope checks therefore improve evidence reuse even when semantic selection and candidate verification remain active. Removing plan-conditioned selection lowers Pass@2 by 3.50 points and Avg. by 2.25 points, with cost rising to \$0.42. Because the Planner continues to revise plans, this result identifies the value of feeding its updated decisions back into evidence selection. Removing both mechanisms produces the largest decrease among the targeted variants: 7.50 Pass@2 points and 4.75 Avg. points, at a cost of \$0.46. Each mechanism improves success both with and without the other; Full also has the lowest cost among these four configurations. Together, the results distinguish two useful decisions in shared-context management: which evidence applies to the recipient, and which applicable evidence serves its updated plan.

C Branch Scalability and Role Complementarity

We examine whether GateTree benefits from additional execution branches and how shared collaboration compares with independent attempts. This study uses the same 200-task SWE-bench Pro subset and DeepSeek backbone as the component ablation, with comparable base execution settings and per-branch budgets. Each branch receives the same budget, so the total available inference budget scales with branch count. We compare success at matched branch or attempt counts and report actual token use separately.

GateTree uses two, four, or eight shared branches, with equal numbers of conservative and exploratory workers: $1 + 1$, $2 + 2$, and $4 + 4$, respectively. GateAgent manages evidence exchange among the branches. The independent baselines run eight attempts for each role, without sharing information between attempts. For each role, we measure cumulative success over the first k attempts, $k = 1, \ldots, 8$: a task is solved if at least one attempt succeeds. GateTree uses the same task-level criterion across its N branches. Token use is the mean per task across the corresponding branches or attempts. Figure 8 compares coverage and token use; Table 4 reports the complete results.

Shared collaboration improves coverage as GateTree scales from two to eight branches. Success rises from 54.5% to 59.0% and 62.5%, corresponding to 109, 118, and 125 solved tasks. The gains decrease from 4.5 to 3.5 percentage points at successive doublings, indicating diminishing returns to additional branches. At matched counts of two, four, and eight, GateTree exceeds both independent baselines. Relative to exploratory repetition, the stronger baseline, its gains are 4.0, 3.5, and 1.5 points, respectively.

Shared collaboration improves both task coverage and token efficiency. At two branches, GateTree uses 0.828 million tokens per task, compared with 1.134 million for two independent exploratory attempts. At eight branches, it solves 125 tasks using 3.309 million tokens per task, compared with 122 tasks and 4.368 million tokens for exploratory repetition. The reductions at matched counts of two, four, and eight are 26.98%, 24.60%, and 24.24%, respectively. Shared heterogeneous execution thus achieves broader coverage with fewer tokens than repeating either role independently at all three matched scales.

The overlap analysis reveals complementary coverage from the two roles. Exploratory attempts solve more tasks overall, while conservative attempts contribute additional successes at every scale. For $k = 8$, the two roles share 111 successes, with eight additional tasks solved only by the conservative

Table 4: Shared-branch and independent-attempt results. C and E denote conservative and exploratory workers. Tokens are mean totals per task in units of 10^4; k denotes independent attempts and N shared branches.

k or N	Method	Roles	Solved	Success	Tokens (10^4)
1	Conservative independent	1C	77/200	38.5%	57.8
1	Exploratory independent	1E	79/200	39.5%	61.3
2	Conservative independent	2C	91/200	45.5%	111.9
2	Exploratory independent	2E	101/200	50.5%	113.4
2	GateTree shared	1C + 1E	109/200	54.5%	82.8
3	Conservative independent	3C	96/200	48.0%	166.6
3	Exploratory independent	3E	106/200	53.0%	164.9
4	Conservative independent	4C	103/200	51.5%	219.3
4	Exploratory independent	4E	111/200	55.5%	219.9
4	GateTree shared	2C + 2E	118/200	59.0%	165.8
5	Conservative independent	5C	110/200	55.0%	274.5
5	Exploratory independent	5E	114/200	57.0%	274.8
6	Conservative independent	6C	113/200	56.5%	326.5
6	Exploratory independent	6E	117/200	58.5%	328.1
7	Conservative independent	7C	115/200	57.5%	380.0
7	Exploratory independent	7E	121/200	60.5%	380.7
8	Conservative independent	8C	119/200	59.5%	436.6
8	Exploratory independent	8E	122/200	61.0%	436.8
8	GateTree shared	4C + 4E	125/200	62.5%	330.9

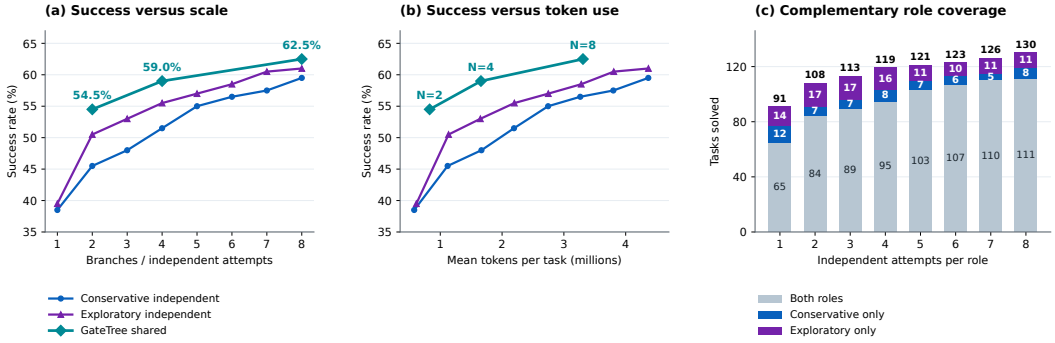

Figure 8: Scaling shared branches and independent attempts on 200 SWE-bench Pro tasks with DeepSeek. (a) Success at matched branch or attempt counts. (b) Success versus mean token use per task. (c) Overlap between tasks solved by the first k conservative attempts and the first k exploratory attempts; bar totals are their union and therefore involve $2k$ independent attempts.

role and eleven only by the exploratory role. Their union covers 130 tasks across sixteen independent attempts. This complementary coverage explains the value of retaining both strategies as branch count increases.

D Failure-Mode Analysis Protocol and Illustrative Cases

We analyze the same 200 SWE-bench Pro tasks for DeLM, AgentFugue, and GateTree under GLM-5.2 and DeepSeek-V4-Pro-0813, yielding 1,200 complete framework runs. GPT-5.6 Sol with maximum reasoning effort in Codex first examines trajectories from 100 tasks per framework and groups recurring problems into four categories. We then apply these definitions to all runs, using one binary judgment per category. The resulting categories and annotations are subsequently reviewed and checked by the authors. Table 5 defines the criteria and gives schematic examples. Tool and environment issues are included for context but excluded from the four modes plotted in Figure 1.

Each judgment covers a complete two-branch run. We mark a mode as present when either branch shows a qualifying problem, even if the run later recovers or the other branch succeeds. Labels may overlap: reusing a stale test result can be an information-governance problem, while failing to recheck the current revision can also be a validation problem. Labels require evidence of the specified behavior in the trajectory and record its occurrence within each outcome group.

Within each model–framework combination, we group runs by final task-level Pass@2 and divide each mode count by the corresponding outcome-group size. For example, information-governance problems occur in $31/96 = 32.29\%$ of successful GLM-5.2/DeLM runs and $48/104 = 46.15\%$ of failed runs. The successful and failed groups therefore have different denominators, and the four percentages need not sum to 100%.

Table 5: Problem categories with schematic recovery and unresolved-failure examples illustrating the labeling criteria.

Mode and criterion	Recovery in a successful run	Unresolved problem in a failed run
Information governance. Unsupported, conflicting, or stale information is shared or reused without adequate evidence or scope checks.	A test result is reused after the patch changes. A later check identifies the revision mismatch, and the corrected candidate passes.	A stale test result is treated as evidence for a revised patch, leaving a regression undetected.
Execution rework. Sharing, implementation, or state reconstruction repeats completed work without meaningful task progress.	Both branches repeat an implementation before one reuses a supported finding and completes the repair.	Repeated handoffs and reimplementation exhaust the budget without producing a coherent, validated candidate.
Validation and termination. Required checks or evidence-based completion are delayed, omitted, or misinterpreted.	Unclear completion evidence triggers redundant checks. The run eventually verifies and submits the valid candidate.	A passing local test is treated as sufficient validation, while required behavior remains unchecked and fails evaluation.
Implementation errors. A candidate follows an incorrect repair direction, modifies the wrong scope, or introduces faulty behavior.	An initial patch addresses a symptom. Execution feedback redirects the repair to the underlying cause, and the final candidate passes.	Workers retain a symptom-level fix based on an incorrect diagnosis, and the submitted candidate fails.
Tool and environment issues. Command, dependency, runner, or workspace faults interrupt execution or evaluation.	Test execution fails because of a setup error. The environment is repaired, and the final candidate passes.	Persistent workspace or dependency errors prevent the team from producing a viable submission.

E Implementation Contract and Scope

Algorithm 1 gives the coordination cycle for two concurrent branches. The Planner dispatches the initial branch plans. When a Worker finishes, GateAgent admits available reports and prepares context for that branch's next planning decision. The revised plan and selected evidence then go to the Worker. The other branch can continue executing during this cycle.

GateAgent admits observations during Worker execution as well as at the planning events shown in the algorithm. Here, J^b is the Planner review capsule, and K^b is the Worker execution capsule refreshed during execution. Delivery checks eligibility, ranks and packs records within the recipient budget, and records receipts. The scope q^b specifies branch, role, phase, and artifact revision; for K^b, it selects the Worker role and implementation phase. Plan checks reject unsupported actions and unchanged failed plans, carrying unmet obligations into the next decision. Submission requires a valid deliverable, requirement coverage, verification on the current candidate, and no unresolved validation failure under the configured checks. The benchmark evaluator separately determines whether the submitted candidate succeeds.

Algorithm 1 GateTree coordination cycle

Input: Task x, resource budget $\mathcal{B}$, delivery budgets W^A, W^B
Output: Separate submissions from branches that satisfy readiness checks

Initialize branches A, B with private states S^b and plans P^b
$\mathcal{A} \leftarrow \{A, B\}$; $\mathcal{M} \leftarrow \emptyset$
Set $K^b, z^b \leftarrow \emptyset$ and $\text{first}^b \leftarrow \text{true}$ for each branch
while resources remain and $\mathcal{A} \neq \emptyset$ **do**
    Wait for an available branch or a worker completion event
    if worker b has completed with output (S', z', R') **then**
        $S^b \leftarrow S'$; $z^b \leftarrow z'$
        Queue report R' for GateAgent and mark b available
    end if
    Select an available branch $b \in \mathcal{A}$
    if first^b **then**
        $\text{first}^b \leftarrow \text{false}$
    else
        $R \leftarrow$ CollectReports
        $\mathcal{M} \leftarrow$ Admit$(\mathcal{M}, R)$
        $J^b \leftarrow$ Deliver$(\mathcal{M}, b, \text{planner}, \text{review}, P^b)$
        $P^b \leftarrow$ UpdatePlan(P^b, J^b, z^b)
        $q^b \leftarrow$ RecipientScope(S^b, P^b)
        $\mathcal{E}^b \leftarrow$ Eligible$(\mathcal{M}, q^b)$
        $\mathcal{L}^b \leftarrow$ Rank$(\mathcal{E}^b, x, P^b, q^b)$
        $K^b \leftarrow$ Pack$(\mathcal{L}^b, W^b)$
        Record delivery receipts for K^b
    end if
    $(a^b, U^b) \leftarrow$ CheckPlan(P^b, K^b, S^b, z^b)
    if $a^b = \text{blocked}$ **then**
        $z^b \leftarrow$ BlockedOutcome(U^b)
    else if $a^b = \text{submit}$ **then**
        if the branch satisfies submission-readiness checks **then**
            Submit the branch's own artifact; $\mathcal{A} \leftarrow \mathcal{A} \setminus \{b\}$
        else
            Record unmet verification obligations in z^b
        end if
    else if $a^b = \text{stop}$ **then**
        $\mathcal{A} \leftarrow \mathcal{A} \setminus \{b\}$
    else
        Dispatch (S^b, P^b, K^b) to worker b and mark it busy
    end if
end while

E.1 Record admission and lifecycle

For software tasks, record management uses explicit artifact revisions and evidence metadata. The rules below specify this implementation; Appendix E.4 describes how reasoning records are managed when executable feedback is unavailable.

Each shared record stores its type, content, evidence references, originating branch, task or artifact scope, repository revision, version, audience, visibility, and lifecycle status. Admission checks type, non-empty content, and metadata. FACT, OBSERVED, and PATCH_SUMMARY records must include evidence references; recognized private-evaluation signals are excluded. Admission establishes the record's declared support and permitted use; subsequent branch reasoning and validation assess its implications for the candidate.

GateAgent reconciles records with matching keys and conflict scope. This scope usually includes the branch and artifact revision. For cross-branch facts and observations, the branch is omitted so reports about the same revision can be compared. Verification reports keep branch scope because they describe checks on one branch's candidate. Exact duplicates update duplicate accounting without creating a new item. Conflicting normalized contents at the same version are retained as disputed records. An accepted newer version supersedes older live versions; an older incoming version is

rejected as stale. Rejected, disputed, and superseded entries remain distinguishable in the store and audit history, but ordinary delivery excludes them. Figure 2 depicts the three possible retained states of a record.

A record version identifies an update to a report. An artifact revision identifies the candidate on which the evidence was obtained. Changing the candidate can make an earlier test result inapplicable even if no newer report has superseded it. An explicitly portable observation may remain useful after an edit. GateAgent checks revision compatibility at delivery time, allowing the same stored record to remain eligible for one recipient while being excluded from another's capsule.

Table 6: Shared-context management: admission and eligibility determine permitted reuse before relevance-based selection.

Decision	Condition examined	Consequence
Admission	Format, declared support, permitted source, and version consistency.	Accept, reject, deduplicate, dispute, or supersede records.
Applicability	Active status; recipient role, phase, audience, branch, and compatible revision scope.	Form the permitted candidate set for this request.
Delivery history	Whether this recipient has already received the record; bounded retention of high-signal evidence.	Suppress redundant delivery without changing validity.
Selection	Relevance to the current task and recipient, complementarity, priority, and the delivery window.	Choose which permitted records to include now.
Execution validation	Coverage and checks on the receiving branch's current candidate.	Determine whether further work or verification is needed before submission.

E.2 Recipient-specific selection and delivery

Figure 9 follows admitted records through Planner review and Worker execution. Each recipient receives a separate capsule, and new outcomes return to the shared store for later decisions.

For each delivery request, GateAgent checks active status, record types allowed for the role and phase, audience, revision compatibility, and branch visibility. Cross-branch delivery also requires permission to share, evidence references, and either matching or explicitly portable revision scope. Peer records may have type FACT, OBSERVED, FAIL, or PATCH_SUMMARY. These checks define the candidate set for relevance-based selection.

Review and execution capsules. The Planner review capsule J^b can combine a peer's supported progress with the current branch's failed check or unmet requirement. The Planner uses this information to choose the next objective. GateAgent then forms K^b for the updated plan, Worker role, execution phase, and current revision. Review emphasizes evidence for choosing the next objective or verification target; execution emphasizes evidence for carrying out that decision. Eligible plan handoffs are prioritized as execution contracts, followed by selected background evidence.

Selection and fallback. After admission and lifecycle updates, the software-task selector receives eligible records, the current task, and the recipient description. It returns an ordered list of existing record IDs and a requested delivery budget, leaving record contents and status unchanged. The budget is clamped to the configured bounds. Eligible plan handoffs precede selected background records. Packing follows that order and skips any item that does not fit, so a smaller later item can still be included. Appendix F gives the selection prompt and response checks.

If the model is unavailable or its output fails validation, deterministic selection orders eligible records by current-task match, plan-handoff status, evidence strength, explicit priority, type priority, version, and recency. It then packs records under the same delivery limit. The fixed 2K study holds this limit

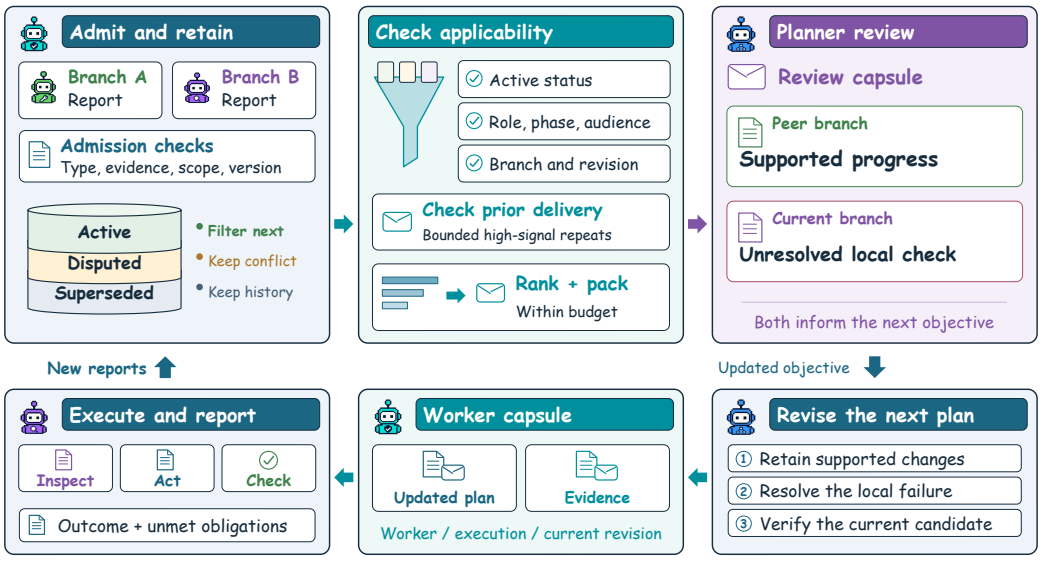

Figure 9: GateAgent in subsequent coordination. Active records undergo recipient and revision checks before review delivery. Supported peer progress and unresolved local checks inform the revised plan; a separate Worker capsule carries the updated task and selected evidence into execution. New reports return to managed context. Disputed and superseded records remain retained but are excluded from ordinary delivery.

constant across selection policies, while the 2K–16K study allows it to vary within those bounds. Eligibility checks apply in both settings and in the fallback path.

Delivery history and budget accounting. Receipts track delivered record sequences separately for each branch, role, and recipient identity. Previously delivered items are normally suppressed, with a reserved part of the window retaining up to three evidence-backed, high-signal records. This keeps a useful current patch summary or failure available across successive decisions. A new record version is considered separately from the earlier version. The receipt ledger tracks recipient exposure, while record status and the task ledger track evidence state and requirement completion.

The final capsule may include selected records, retained high-signal evidence, and task-state information. Planner review can include the task ledger; Worker execution omits it. An unchanged task-state block is suppressed if the recipient has already seen it. After assembling the capsule, the renderer removes trailing records as needed and applies a final text-length check against the delivery window. This check uses the configured token counter. The context study uses the DeepSeek tokenizer separately during source construction. Delivery-window accounting covers the assembled capsule; total inference accounting also includes private history, prompt instructions, and subsequent calls.

E.3 From shared evidence to revised branch plans

At initialization, there is no shared execution evidence, so the Planner can dispatch the initial plans directly. After a Worker returns, GateAgent admits available reports and supplies a review capsule. The Planner revises that branch's plan, and GateAgent prepares an execution capsule for the new task. Each branch advances as its own outcomes become available, retaining its candidate and submission throughout the cycle.

Plans specify an objective, assumptions with evidence references, intended actions, expected evidence, and success criteria. Requirements stated in the task are part of the contract. Inferred file locations, behaviors, and compatibility claims may require supporting evidence. Plan review checks required fields and cited references. An unsupported critical premise must be revised or tested through a

bounded read-only probe with an explicit inspection scope. Both branch strategies follow this rule, connecting critical assumptions to evidence that the Worker can inspect and assess.

After an incomplete result, the next proposal must address failed checks, invalid artifacts, remaining requirements, and new evidence while preserving useful completed work. To detect replay, the implementation compares plan signatures over the objective, assumption contents, intended steps, expected evidence, success criteria, and probe constraints. The signature excludes record IDs and context-version identifiers, so metadata changes preserve the replay match. This structured comparison is complemented by the Planner instruction to revise the underlying hypothesis when evidence contradicts it.

For example, branch A may report a supported change to a shared interface while branch B still has a failing boundary test. An eligible summary from A can enter B's Planner review alongside the local failure. The revised plan can retain B's valid changes, target the unresolved behavior, and require a check on the current candidate. The Worker receives the task and applicable evidence, executes locally, and returns an outcome. The peer report directs inspection and verification toward the unresolved behavior in B's own candidate.

Submission review checks the Worker's completion report against current-candidate evidence and requirement coverage. Unresolved failures and missing checks return to planning; a ready branch submits its own candidate for evaluation. Task-ledger states such as running, validating, and done describe execution progress, whereas active, disputed, and superseded describe evidence records. The two state spaces are maintained separately. Branch-local regulators apply revised evidence to each branch's own plan history, preserving distinct solution paths.

E.4 Shared evidence without executable feedback: HLE

Text-only HLE presents the same coordination problem in a reasoning setting: a conclusion useful to one branch may depend on an assumption that another branch has revised. GateTree therefore manages the applicability of shared arguments as the two answers evolve. Each branch retains its own draft and reasoning history, while GateAgent uses derivations and critiques to update shared records and prepare context for subsequent reasoning. No search, code execution, or test feedback is available.

The original question supplies the common task conditions. Branch records retain their source and supporting argument, distinguishing derived conclusions from tentative hypotheses. When GateAgent assesses a critique as conflicting with an existing claim, it marks that claim disputed; an accepted revision supersedes the earlier claim. The distinction matters for planning. A dispute leaves a premise to resolve; a replacement allows the branch to proceed from the revised claim. Source and format checks preserve attribution, while the LLM assesses the arguments supporting these updates.

GateAgent selects context separately for each branch using its current draft, strategy, and unresolved reasoning needs. If a peer's critique challenges a premise in the receiving branch's draft, the relevant argument informs the next plan: reconsider that premise before extending the derivation. An accepted replacement instead provides a revised basis for continued reasoning. The branch combines this context with its own draft to update its plan and answer. Thus, lifecycle management changes which findings remain usable, and recipient-specific delivery connects those changes to the next reasoning step. The capsules may overlap when evidence is useful to both branches, while their candidate answers remain separate.

Branches produce an explanation, answer, and confidence report within the configured revision budget. Source checks establish attribution, and LLM review assesses arguments to guide revision. Correctness is evaluated against the held-out benchmark reference, separately from this internal review. HLE thus tests evidence-guided collaboration using derivations and critiques in place of executable feedback. Table 7 summarizes how changes in shared reasoning inform branch decisions.

E.5 Token-to-cost conversion

We convert token usage to API-equivalent cost using fixed input and output rates per million tokens: CNY 3 and CNY 6 for DeepSeek-V4-Pro-0813, and CNY 8 and CNY 28 for GLM-5.2, respectively. Input tokens use the cache-miss rate. For input and output token counts T_{in} and T_{out}, the

Table 7: From shared reasoning to branch decisions in HLE.

Shared information	GateAgent operation	Role in the next decision
A critique conflicts with an earlier claim	Assess the conflict, mark the contested claim disputed, and select an eligible supporting critique.	Reconsider the affected premise before extending the argument.
An accepted revision replaces a claim	Mark the earlier record superseded and select the applicable revision.	Continue from the revised claim rather than its outdated predecessor.
A peer derivation addresses a remaining question	Select the source-attributed argument for the branch that needs it.	Incorporate relevant progress into the branch's own plan and answer.

corresponding cost in USD is

$$C_{\mathrm{USD}} = \frac{p_{\mathrm{in}} T_{\mathrm{in}} + p_{\mathrm{out}} T_{\mathrm{out}}}{10^6 \times 7.2},$$

where p_{in} and p_{out} are the model-specific rates in CNY, and the fixed exchange rate is CNY 7.2 per USD.

F Role Prompts and Response Formats

The following boxes reproduce selected SWE-bench Pro implementation instructions. Placeholders replace runtime values; repeated task text and command documentation are omitted. Both Workers use the same execution and reporting interface, with different starting strategies and verification priorities.

F.1 Planner: evidence-conditioned decisions

The Planner receives the task contract, branch profile, remaining budget, latest branch outcome, adaptive plan state, and a fresh review capsule. The instruction blocks below give its evidence, replanning, and submission rules. Repository identifiers and runtime history are supplied at each call. The Worker executes the delegated task in its branch environment.

Planner: shared evidence and replanning rules

Shared evidence
Use FACT/OBSERVED as evidence, TRIED as history, and FAIL to avoid repeating a known dead end. Treat PATCH_SUMMARY as a candidate until locally verified. When delegating near a recorded FAIL, state the alternative explicitly. Active CLAIMs identify duplicate work.

Dynamic replanning
next=REVISE: do not replay the same plan/task. Change the objective or task_instruction to address the recorded outcome, issues, and new Gate evidence.

next=SUBMIT_OR_EXTEND: submit only when Worker self-test evidence covers the source fix; otherwise delegate a specifically scoped missing check.

Every delegate_task must be the current best next task, not a stale continuation. Make each approach falsifiable; contrary evidence must change the hypothesis, not merely its wording or file list.

Submission readiness
Require a non-empty valid deliverable patch; generated/transient artifacts and diff-hygiene failures must be corrected before submission. Require current-revision execution evidence mapped to every externally observable requirement, plus a relevant boundary/negative case. A successful but unrelated command is not submission evidence.

Preserve the requirement IDs attached to a delegation. Submit only when every ID is explicitly covered and mapped to an actually run behavioral command/result. Treat any unresolved check failure or stated unmet requirement as a replan signal.

Replan from structured evidence: failure_type, failed_checks, invalid_files, requirements_remaining, blocker, and recommended_next_action. Do not replay an unchanged delegation after an incomplete or invalid result.

Planner: assumptions, bounded probes, and output constraints

Return EXACTLY ONE JSON object and NOTHING else. "action" MUST be exactly "delegate_task" or "submit". Include a top-level "plan" object with: objective, assumptions, expected_evidence, and success_criteria. Each assumption is {"claim": string, "evidence_ids": [string], "probe": boolean}.

Both branches are complete solvers. Put uncertain file locations or implementation details in task_instruction as things for Worker to verify before editing; do not promote them to critical plan assumptions without Gate evidence.

Requirements and Interface statements copied directly from TASK are contract, not assumptions: omit them from assumptions. Any extra file-location, behavior, or compatibility claim inferred by Planner is an assumption and must obey the rules below.

If any unsupported assumption has "probe": true, the plan MUST also include "read_only": true and a non-empty "probe_scope" listing the exact files, symbols, tests, or commands the delegation may inspect. The delegated task must not edit files. Missing or unbounded probe scope is invalid. Plans that edit code must not rely on an unsupported probe assumption.

task_instruction may repeat TASK contract and Gate-backed facts. It must not turn an unsupported Planner inference into an edit requirement or ask Worker to modify tests. Cite only IDs that actually appear in the Gate context. Never copy placeholder text.

Evidence-backed Gate FACT/OBSERVED takes precedence over unsupported prior reasoning. If it conflicts with the current plan, verify the cited scope or revise the task instead of silently ignoring it. If an evidence-gathering delegation needs no premise, use "assumptions": [].

The Planner returns the following response structure. task_instruction is required for delegation and reason for submission; probe-specific fields apply only when needed.

Planner: response format

```
{
  "action": "delegate_task | submit",
  "reasoning": "<evidence-based decision>",
  "plan": {
    "objective": "<current objective>",
    "assumptions": [
      {"claim": "...", "evidence_ids": [], "probe": false}
    ],
    "expected_evidence": ["..."], "success_criteria": ["..."]
  },
  "params": {
    "task_instruction": "...", "context": "...", "reason": "..."
  }
}
```

F.2 WORKERS: DIFFERENTIATED SEARCH, COMMON COMPLETION CRITERIA

The complete strategy prefixes are reproduced below. Each branch produces and verifies its own candidate, combining independent problem solving with complementary search priorities.

Worker strategy prefixes: two complete solvers

A. Conservative

INTERNAL BRANCH: CONSERVATIVE COMPLETE SOLVER. Independently produce and submit a working patch. Prefer a minimal reproduction, verified facts, the smallest sound source change, and focused tests. State one causal hypothesis and its cheapest falsifying check before broadening scope.

B. Exploratory

INTERNAL BRANCH: CONTRACT-COVERAGE COMPLETE SOLVER. Independently produce and submit a working patch. Start from externally observable requirements, identify integration and boundary risks, then implement the smallest sound change and verify the affected contract. Use the same repository architecture. State a causal hypothesis distinct from the obvious/local path and its cheapest falsifying check before editing.

Worker: requirement coverage and verification contract

Keep a short checklist of externally observable requirements through implementation and verification. Inspect direct consumers of changed shared interfaces or schemas.

Before declaring done, leave a real patch and map every item to an actually run behavioral check. Add a boundary/negative check and, when relevant, an adjacent build/static/regression check. Run git diff –check when available; exclude generated, cache, editor, compiled, and conflict artifacts.

For Gate-counted verification, run the test/build/static command directly, with only an optional leading VAR=value. Do not hide its exit status in bash -c, a pipe to tail, ; echo $?, or ||. For a custom temporary reproduction, run verify <single focused command> so Gate can record its exit status.

Keep PATCH_SUMMARY under 600 characters and put bookkeeping first: covered=... | remaining=... | evidence=... | risk=.... If delegation has IDs (R1, R2, ...), report every ID, e.g., covered=R1,R2 and evidence=R1:<command/result>; R2:<command/result>. This must be a requirement-to-command mapping using current-revision evidence; predicted, skipped, irrelevant, or stale results do not count.

Worker responses contain a brief action rationale in DISCUSSION, an optional typed NOTE, and one command per COMMAND block, followed by END_OF_RESPONSE with no trailing text. Dependent actions wait for feedback. Notes distinguish facts, attempted approaches, observations, failures, claims, and patch summaries. A PATCH_SUMMARY records the affected files, repair idea, evidence, covered and remaining requirements, and risks; evidence links requirements to executed commands and results. A finish command must be the only command in its response and distinguishes verified completion (done) from progress with unmet requirements (partial).

F.3 GATEAGENT: CONTEXT SELECTION

The prompt below implements recipient-specific selection for software tasks. It receives records admitted under the rules in Appendix E.1 and identifies those needed for the recipient's next decision. Runtime fields specify the task, recipient, candidate records, and configured budget bounds.

GateAgent: semantic selection prompt

You are the semantic context-routing Gate for a software-engineering agent. The records below already passed deterministic security, branch, revision, audience, and evidence checks. Select the smallest sufficient set for the current recipient. Prefer verified, current, complementary evidence; retain failures that prevent repeated mistakes; remove redundant or low-value history. Never solve the task, rewrite evidence, or invent IDs. Return JSON only with schema:

{"selected_ids":["existing-id"],"budget_tokens":1234}.

budget_tokens must be between *floor* and *ceiling*; selected_ids may be empty only when none of the records helps.

Runtime suffix:
TASK: current task
RECIPIENT: role, phase, branch, repository revision
SAFE CANDIDATE RECORDS: ID, type, branch, priority, producer role, message kind, evidence IDs, and content.

GateAgent: runtime candidate format

```
TASK
<current task or delegated objective>
RECIPIENT
role=<role> phase=<phase> branch=<branch>
repo_revision=<current revision>
SAFE CANDIDATE RECORDS
[
  {
    "id": "<existing record ID>", "type": "<record type>",
    "branch": "<originating branch>", "priority": <priority>,
    "producer_role": "<producer role>",
    "message_kind": "<message kind>",
    "evidence_ids": ["<existing evidence reference>"],
    "content": "<retained record content>"
  }
]
```

Each object is an eligible candidate. Its branch field identifies the record's origin; the recipient is specified separately. Returned record IDs select items for delivery, while evidence IDs within those items refer to supporting observations. Record contents and lifecycle states are maintained separately; the selection response specifies only which records to deliver and their token budget.

The response is validated before packing. Unknown IDs are rejected, repeated IDs are deduplicated, and the requested budget is clamped to its configured bounds. A malformed response receives one strict-format retry. Eligible plan handoffs precede selected background records, subject to the delivery limit. If parsing, validation, or the model call still fails, deterministic packing selects from the same eligible set.