REPOMAS: SOLVING PROGRESSIVELY SPECIFIED TASKS WITH ISSUE-DRIVEN MULTI-AGENT SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-based multi-agent systems (MASs) have shown strong potential for solving complex tasks, but most assume that task requirements are sufficiently specified before execution. In practice, user requests are often incomplete, and additional requirements may only become clear during reasoning, tool use, or execution. We refer to such problems as *progressively specified tasks*.

To systematically study this setting, we introduce **ProgSpec**, a benchmark that evaluates final outputs against requirements explicitly stated in the initial request and additional requirements supported by the available task evidence. We further propose **RepoMAS**, an issue-driven multi-agent framework inspired by open-source project management. RepoMAS records newly discovered requirements, conflicts, and failures as structured Issues and uses them to revise the task specification and execution structure during problem solving.

Across ProgSpec and five existing benchmarks, RepoMAS achieves the best performance. Further analyses show that its issue-driven revision and repository maintenance mechanisms consistently contribute to performance. These results highlight the importance of allowing MASs to revise not only how a task is solved, but also revise their explicit representation of task requirements during execution.[1]

1 INTRODUCTION

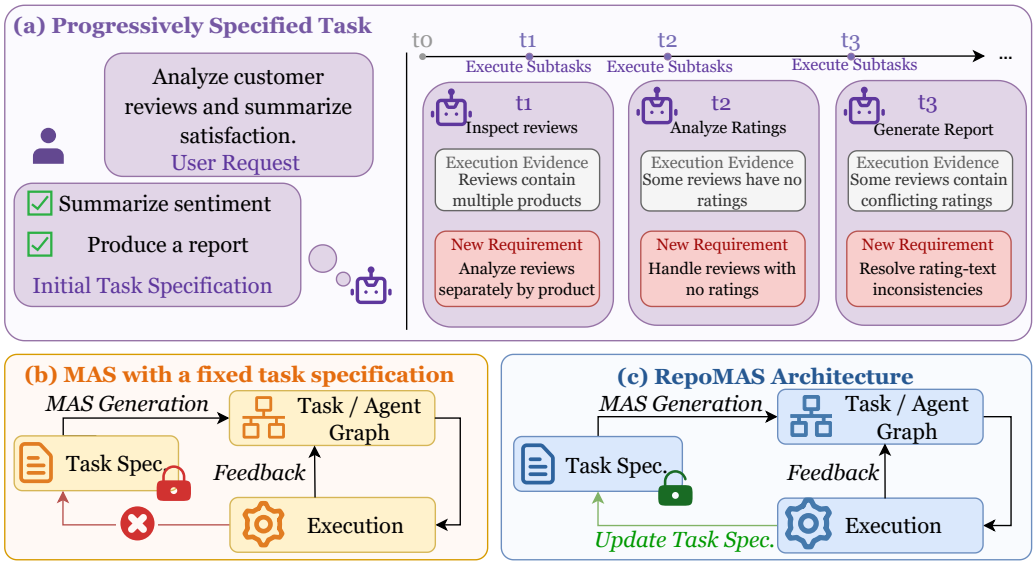


Figure 1: **Overview of progressively specified tasks and RepoMAS.** Task requirements may emerge gradually from execution evidence rather than being fully specified in the initial request. RepoMAS records these newly discovered requirements and revises the task specification and execution structure accordingly.

[1] Our code and data will be available once the paper is accepted.

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.

LLM-based multi-agent systems (MASs) have become a promising paradigm for solving complex tasks. Instead of relying on a single model to complete an entire task, MASs decompose complex problems into subtasks and assign them to specialized agents. By organizing agents into specialized roles and enabling them to collaborate, these systems can handle complex tasks more flexibly and effectively (Guo et al., 2024; Hong et al., 2024; Chen et al., 2024).

While existing MASs have made substantial progress, most MASs implicitly assume that the task is sufficiently specified before execution begins, as shown in Figure 1(b). Users usually describe an overall goal in a few sentences, but they may not be able to specify every requirement in advance. Some specifications remain latent until a tentative solution makes their absence visible, while others become explicit only after the system retrieves evidence, invokes tools, or interacts with the environment. As these issues are discovered, both the solution and the system's understanding of the task must be revised accordingly. As illustrated in Figure 1(a), some task requirements become visible only during execution. We refer to such tasks as *progressively specified tasks*.

To systematically study this setting, we introduce **ProgSpec**, a benchmark for progressively specified tasks. It evaluates whether a system can identify and handle specifications that are missing from the initial request and discovered during problem solving. To address progressively specified tasks, we propose **RepoMAS**, an issue-driven multi-agent framework that allows the task specification to be refined during problem solving. Its design is inspired by the workflow of open-source project management, where newly discovered problems are recorded as Issues, addressed through targeted revisions, and incorporated only after validation.

Given an initial user request, RepoMAS first constructs a Task Draft that reflects its current understanding of the task. As execution proceeds, retrieved evidence, tool use, and intermediate results may reveal previously unspecified requirements. RepoMAS records these findings as structured Issues, which capture what has been discovered and what needs to be revised. These Issues first revise the Task Draft and trigger corresponding updates to the task structure and agent organization. After revising the repository state, RepoMAS locally re-executes only the affected tasks. This iterative process allows RepoMAS to start from an incomplete request and progressively refine its understanding. A simplified overview of RepoMAS is shown in Figure 1(c).

In summary, this work makes three main contributions:

1. We formulate progressively specified tasks and introduce ProgSpec, a benchmark for evaluating whether systems satisfy requirements that are not explicitly stated in the initial request.

2. We propose RepoMAS, an issue-driven multi-agent framework that turns execution feedback into structured Issues, which guide revisions to the current task specification.

3. With GPT-4o-mini as the backbone, RepoMAS outperforms the evaluated baselines on ProgSpec and five established benchmarks. Experiments with additional backbones further demonstrate its applicability across different model families and tasks.

2 Related Work

2.1 LLM-based Multi-Agent Systems

LLM-based multi-agent systems solve complex tasks through role specialization, communication, and coordinated workflows (Guo et al., 2024; He et al., 2025; Tran et al., 2025). Representative systems, such as CAMEL (Li et al., 2023), DyLAN (Liu et al., 2023), and AgentVerse (Chen et al., 2024), study how multiple agents collaborate through natural-language interaction.

Later work further treats multi-agent organization as a structure that can be explicitly designed and optimized. MacNet (Qian et al., 2025) studies large-scale collaboration through communication graphs, while AgentPrune (Zhang et al., 2025b) removes redundant connections to reduce communication cost. Together, these studies examine how communication structure affects collaboration and its computational cost.

2.2 Dynamic Task Planning and Agent Collaboration

Recent works move beyond fixed task plans toward task-dependent and execution-time adaptation. AutoAgents (Chen et al., 2023) jointly constructs task-specific agents and execution plans, while MaAS (Zhang et al., 2025a) selects multi-agent architectures according to task complexity. Other approaches (Yu et al., 2025; Wang et al., 2026b; Xu & Tai, 2026; Li et al., 2026) revise task decomposition, dependencies, or execution structures based on intermediate results and failures.

Agent organization can also be adapted during execution. Evolving Orchestration (Dang et al., 2026) and AgentConductor (Wang et al., 2026a) adjust agent activation or interaction structures, while other methods (Ruan et al., 2026; Wang et al., 2026c; Xu et al., 2026) further revise agent roles, capabilities, or communication patterns.

RepoMAS maintains explicit requirement–task links. Newly identified constraints can trigger coordinated revisions to the task specification and execution structures. These revisions are managed through structured Issues and validated patches.

2.3 Benchmarks for Agents and Incomplete Specifications

Table 1: **Comparison between ProgSpec and related agent benchmarks.** Autonomous requirement recovery denotes recovery from reasoning or execution evidence rather than requirements supplied through user clarification.

Benchmark	Incomplete Specifications	Autonomous Requirement Recovery	Multi-Domain
AgentBench (Liu et al., 2024)	✘	✘	✔
GAIA (Mialon et al., 2024)	✘	✘	✔
MultiAgentBench (Zhu et al., 2025)	✘	✘	✔
SWE-Together (Wu et al., 2026)	✔	✘	✘
ICAE-Bench (Peng et al., 2026)	✔	✘	✘
SpecBench (Hamblin et al., 2026)	✔	✔	✘
ProgSpec (Ours)	✔	✔	✔

General-purpose benchmarks such as AgentBench (Liu et al., 2024) and GAIA (Mialon et al., 2024) evaluate agent capabilities across diverse environments, while MultiAgentBench (Zhu et al., 2025) focuses more directly on multi-agent coordination and communication. Other benchmarks further study agent architectures, team sizes, and collaboration protocols (Grötschla et al., 2025; Zhang et al., 2026; Yang et al., 2026). These benchmarks primarily evaluate capabilities other than autonomous recovery of missing requirements from execution evidence.

More recent benchmarks consider incomplete specifications. SpecBench (Hamblin et al., 2026) evaluates whether software agents can identify omissions, ambiguities, inconsistencies, and incorrect assumptions in software design proposals. SWE-Together (Wu et al., 2026) and ICAE-Bench (Peng et al., 2026) introduce additional requirements through user interaction during coding tasks. ProgSpec complements specification-level benchmarks by evaluating requirement satisfaction across Coding, Mathematics, and document-grounded QA, with evidence drawn from the initial context and subsequent problem solving. Table 1 summarizes these differences.

3 ProgSpec: Progressively Specified Tasks Benchmark

3.1 Progressively Specified Tasks

We define **progressively specified tasks** as tasks where the initial user request does not explicitly state all requirements needed for successful completion. Given a request q, let $\mathcal{R}^*$ denote an idealized, fixed, evidence-grounded requirement set, and let $\mathcal{R}_{\text{vis}} \subseteq \mathcal{R}^*$ denote those explicitly stated in the initial request. In ProgSpec, the idealized set is approximated by a human-constructed reference set. Throughout the benchmark and evaluation sections, $\mathcal{R}^*$ denotes this annotated reference set.

Some missing requirements may be inferred immediately, while others only become clear through reasoning, retrieved evidence, tool use, or execution feedback. Let $\hat{\mathcal{R}}_t$ denote the requirements identified by the system at repository version t. As execution proceeds, the task specification is

progressively refined from $\hat{\mathcal{R}}_0$ to $\hat{\mathcal{R}}_1, \ldots, \hat{\mathcal{R}}_T$ as new requirements are discovered. Unlike tasks with a complete specification upfront, the system must solve the task and update its understanding of the requirements as new task specification emerges.

3.2 Benchmark Construction

We construct **ProgSpec** to evaluate whether a multi-agent system can satisfy requirements that are not fully stated in the initial request. Each instance contains an initial request, an annotated reference requirement set $\mathcal{R}^*$, and annotations describing how directly each requirement can be obtained from the initial task input.

Data collection. ProgSpec covers Coding, Mathematics, and document-grounded QA. For **Coding**, we collect repository-grounded implementation-planning tasks from public GitHub projects, using Issues, pull requests, tests, and repository history as annotation evidence for missing requirements. For **Mathematics**, we select 60 problems from MATH (Hendrycks et al., 2021) and 30 from the text-only subset of OlympiadBench (He et al., 2024). For **QA**, we construct 90 document-grounded tasks from MultiDoc2Dial (Feng et al., 2021), where complete answers may require identifying additional conditions, exceptions, or constraints from the provided documents. Detailed construction procedures are provided in Appendix B.

Specification annotation. We decompose each task into atomic, independently verifiable requirements and assign them to three levels based on how directly they can be obtained from the initial task input. L1 requirements are explicitly stated, L2 requirements can be inferred from the request or standard domain knowledge, and L3 requirements are typically discovered during execution by inspecting task-specific evidence, intermediate results, tool feedback, or verification. The level depends on how directly a requirement is specified, not on when the system discovers it. Appendix C provides the annotation guidelines, and Appendix G reports the independent human validation.

3.3 Data Statistics

ProgSpec contains 270 tasks, with 90 tasks in each domain. In the standard setting, systems receive the public task input, including a repository snapshot or document context where applicable, while requirement annotations and reference answers are withheld. As shown in Table 2, the benchmark contains 1,551 atomic requirements, averaging 5.74 per task, of which 46.2% are L3 requirements.

Table 2: **Statistics of ProgSpec.** L1, L2, and L3 denote explicit, implicit, and latent requirements, respectively.

Domain	Tasks	Reqs.	L1	L2	L3	Reqs./task
Coding	90	539	90	164	285	5.99
Math	90	530	180	213	137	5.89
QA	90	482	90	97	295	5.36
Total	270	1,551	360	474	717	5.74

Evaluation. We use weighted requirement coverage as the primary metric:

$$\text{Coverage} = 100 \times \frac{\sum_{r_i \in \mathcal{S}} w_i}{\sum_{r_i \in \mathcal{R}^*} w_i}, \tag{1}$$

where $\mathcal{S}$ is the set of satisfied requirements and $w_i \in \{1, 2, 3\}$ denotes the weight of L1–L3 requirements. This weighting emphasizes the benchmark's focus on satisfying implicit and latent requirements. Appendix D gives the complete aggregation and satisfaction criteria.

4 RepoMAS: An Issue-Driven Multi-Agent System for Progressively Specified Tasks

RepoMAS organizes multi-agent task solving around a repository-based workflow inspired by open-source development. Instead of treating the initial task plan as fixed, RepoMAS maintains a shared task state that can be inspected and revised throughout execution. Figure 2 illustrates the overall process.

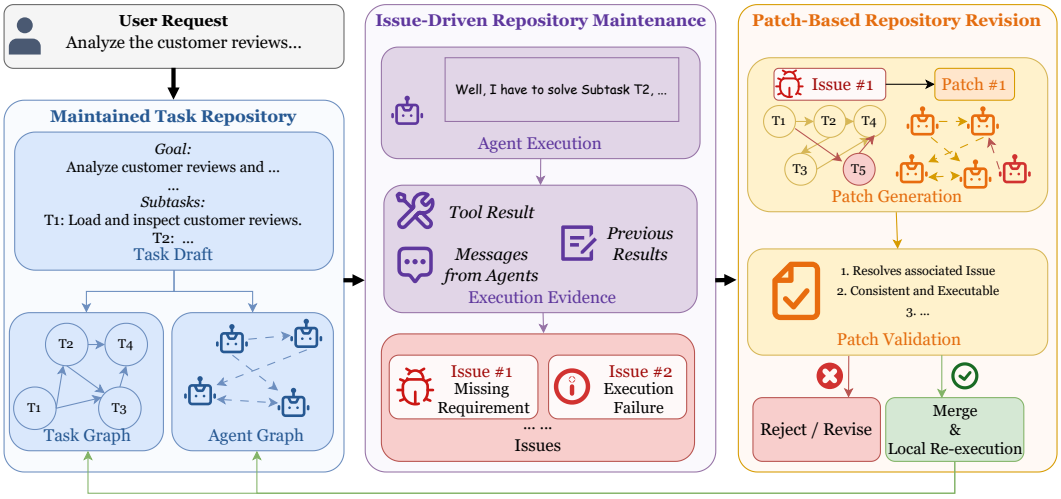


Figure 2: **An overview of RepoMAS**. RepoMAS maintains the task specification and execution structures in a shared task repository. Execution evidence is converted into structured Issues, which drive patch-based revisions to the repository. Candidate patches are validated before merging, and accepted revisions trigger local re-execution of only the affected tasks.

4.1 Repository State Initialization

At repository version t, RepoMAS maintains a **Task Draft** $\mathcal{D}_t = (\hat{\mathcal{R}}_t, \mathcal{O}_t, \mathcal{V}_t)$, which records the currently known requirements, expected outputs, and verification criteria. The draft can be updated as new requirements are discovered.

RepoMAS then decomposes the draft into a directed acyclic **Task Graph** $\mathcal{H}_t = (V_t^H, E_t^H)$. Each node represents a subtask, while edges represent execution dependencies. Each subtask is also linked to the requirements it is responsible for. These links are used to locate the affected subtasks when the task specification changes.

RepoMAS also maintains an **Agent Graph** $\mathcal{G}_t = (V_t^G, E_t^G)$, which describes the agents in the current workflow and their communication structure. Each subtask is assigned to one or more agents through a task-agent mapping $\mathcal{M}_t$.

The repository therefore keeps both the current task structure and the current execution structure, together with the Issues discovered during execution $\mathcal{I}_t$ and the generated artifacts $\mathcal{A}_t$:

$$\mathcal{S}_t = (\mathcal{D}_t, \mathcal{H}_t, \mathcal{G}_t, \mathcal{M}_t, \mathcal{I}_t, \mathcal{A}_t)\,. \tag{2}$$

4.2 Issue-Driven Repository Maintenance

During execution, feedback may reveal problems in the current task specification or workflow. RepoMAS records each finding as a structured **Issue** before proposing a repair. Related Issues are grouped into a **repair branch**, where candidate revisions remain isolated from the accepted repository state until they are validated.

Each Issue records the problem and the evidence needed to locate and verify it, which can be modeled as

$$I_i = (\tau_i, \kappa_i, \phi_i, e_i), \tag{3}$$

where τ_i denotes the issue type, κ_i the affected repository component, ϕ_i the problem description, and e_i the supporting evidence.

For a repair branch B, a candidate patch specifies changes to the repository components:

$$P_B = (\Delta\mathcal{D}_B, \Delta\mathcal{H}_B, \Delta\mathcal{G}_B, \Delta\mathcal{M}_B)\,. \tag{4}$$

When an Issue reveals a missing or incorrect requirement, RepoMAS revises the Task Draft and updates the affected execution structures accordingly. Issues concerning execution failures may instead require changes to the affected tasks or agent assignments without changing the Task Draft.

4.3 Patch Validation, Merge, and Re-execution

Before merging a candidate patch, a validator reviews it against the current repository state, the target Issues, and their supporting evidence. The review assesses whether the proposed changes address the target Issues and remain consistent with the task requirements. Rejected candidates may be revised, while accepted patches are merged into the repository. The effects of an accepted patch are subsequently checked through re-execution of the affected tasks.

Because a patch may invalidate previous results, RepoMAS re-executes only the affected part of the Task Graph. Let $C(P_B)$ denote the tasks directly affected by the patch, while artifacts from removed tasks are invalidated. Unaffected results are preserved for reuse. The re-execution set is

$$\mathcal{V}_{\mathrm{re}} = C(P_B) \cup \mathrm{Desc}_{\mathcal{H}_{t+1}}\left(C(P_B)\right), \tag{5}$$

which includes the directly affected nodes and their downstream dependent tasks in the updated graph. Outputs outside $\mathcal{V}_{\mathrm{re}}$ are reused only when their inputs, assigned requirements, and producing agents remain unchanged.

The accepted-patch path for a repair branch can be summarized as

$$\mathcal{S}_t \xrightarrow{\text{Execute}} \mathcal{I}_t^{\mathrm{new}} \xrightarrow{\text{Branch/Patch}} P_B \xrightarrow{\text{Review/Merge/Local Re-execution}} \mathcal{S}_{t+1}. \tag{6}$$

Here $\mathcal{I}_t^{\mathrm{new}}$ denotes newly recorded Issues, rather than the full Issue history in $\mathcal{S}_t$. A rejected patch remains outside the accepted state and may be revised. Re-execution may reveal further requirements or problems, which are again recorded as Issues. The process continues until no further revision is needed or the execution budget is reached.

5 Experiments

5.1 Experimental Setup

Benchmarks. We evaluate RepoMAS on ProgSpec and 5 conventional benchmarks. Following Meta-Agent (Xu & Tai, 2026), we evaluate on the AFlow (Zhang et al., 2025c) test splits of HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), HotpotQA (Yang et al., 2018), DROP (Dua et al., 2019), and GSM8K (Cobbe et al., 2021). These benchmarks provide a broader evaluation of RepoMAS on conventional coding, reasoning, and QA tasks.

Baselines. We compare RepoMAS with GPT-4o-mini as a single-agent baseline and seven representative multi-agent systems, including MetaGPT (Hong et al., 2024), AgentCoder (Huang et al., 2023), G-Designer (Zhang et al., 2024), AFlow (Zhang et al., 2025c), Meta-Agent (Xu & Tai, 2026), TacoMAS (Xu et al., 2026), and LATTE (Mieczkowski et al., 2026). For a fair comparison, all methods use GPT-4o-mini as the backbone and share the same task inputs, tool environment, and evaluation protocol, while some baseline results are taken directly from their original papers. Appendix H provides additional details on the experimental setup and statistical significance tests.

Metrics. On ProgSpec, we report weighted requirement coverage as the primary metric, which is defined in Appendix D. On the standard benchmarks, we report pass@1 for HumanEval and MBPP, solve rate for GSM8K, and F1 for HotpotQA and DROP.

5.2 Main Results

Table 3 reports the main results with GPT-4o-mini as the backbone. RepoMAS achieves the highest score across all six benchmarks. On ProgSpec, RepoMAS achieves 44.7 weighted requirement coverage, outperforming Meta-Agent by 1.8 points and GPT-4o-mini by 6.4 points. AFlow and Meta-Agent reach 39.9 and 42.9, respectively, while several MASs perform below the single-agent baseline. These results show that RepoMAS is more effective at handling progressively specified tasks. Appendix A presents three RepoMAS cases. Additional results with different backbone models are provided in Appendix F.

Table 3: **Main results with GPT-4o-mini on six benchmarks.** We report pass@1 for HumanEval and MBPP, F1 for HotpotQA and DROP, solve rate for GSM8K, and weighted requirement coverage for ProgSpec. Bold marks the best reported result in each column. * indicates results are reported in the original paper. The overall performance trend is statistically significant ($p = 0.024$).

Method	HumanEval	HotpotQA	DROP	GSM8K	MBPP	ProgSpec
GPT-4o-mini	87.0	68.1	68.3	92.7	71.8	38.3
MetaGPT (Hong et al., 2024)	87.0	67.3	77.3	82.8	66.6	8.9
AgentCoder (Huang et al., 2023)	82.4	54.9	82.0	88.5	59.2	12.8
G-Designer (Zhang et al., 2024)	88.6	68.1	71.5	95.1	71.3	11.1
AFlow (Zhang et al., 2025c)	94.7	73.5	80.6	93.5	83.4	39.9
Meta-Agent (Xu & Tai, 2026)	96.0*	69.5*	82.7*	93.7*	84.6*	42.9
TacoMAS (Xu et al., 2026)	87.8	47.4	36.4	88.1	70.1	16.3
LATTE (Mieczkowski et al., 2026)	90.8	60.4	83.0	86.1	68.3	9.4
RepoMAS (Ours)	**99.2**	**75.5**	**86.3**	**95.8**	**87.1**	**44.7**

6 Analysis

6.1 Ablation Study

We ablate four core components of RepoMAS using GPT-4o-mini on ProgSpec, HumanEval, and HotpotQA. *w/o Structured Issue* replaces the structured Issue representation with a free-form record, while keeping the Issue stage itself. *w/o Patch Validation* removes the validation step before a candidate patch is merged. *w/o Structural Revision* allows the Task Draft to be updated but keeps the task and agent structures fixed. *w/o Local Re-execution* disables affected-subgraph re-execution and reruns the workflow without this locality mechanism.

Table 4: **Ablation study of RepoMAS.** Each variant removes one major component from the full framework.

Setting	ProgSpec	HumanEval	HotpotQA
RepoMAS	**44.7**	**99.2**	**75.5**
w/o Structured Issue	30.8	84.7	68.5
w/o Patch Validation	33.4	84.0	59.4
w/o Structural Revision	30.3	87.8	56.4
w/o Local Re-execution	32.4	84.7	69.7

Table 4 shows that removing any component reduces performance across all three benchmarks. Structural Revision has the largest effect on ProgSpec and HotpotQA. Patch Validation causes the largest drop on HumanEval, highlighting the importance of reviewing candidate revisions before they are merged. Removing structured Issues also leads to substantial degradation, showing the benefit of explicitly organizing execution feedback before repair. Local Re-execution also improves the performance, although its effect is smaller on HotpotQA.

6.2 Effect of Structured Issue Representation

We study how the representation of execution feedback affects RepoMAS using GPT-4o-mini on ProgSpec, HumanEval, and HotpotQA. *Direct Revision* generates a patch directly from execution feedback without creating an Issue. *Free-form Issue* first summarizes the problem in natural language before patch generation. *Structured Issue w/o Evidence* uses the predefined Issue fields but omits supporting evidence. *Full Structured Issue* uses the complete RepoMAS representation, where each Issue is structured and grounded in evidence. All other components remain unchanged.

Table 5: **Effect of different Issue representations.** All variants differ only in how execution feedback is represented before patch generation.

Issue Representation	ProgSpec	HumanEval	HotpotQA	Avg.
Direct Revision	30.2	90.8	68.5	63.2
Free-form Issue	30.8	84.7	68.5	61.4
Structured Issue w/o Evidence	27.5	90.8	69.0	62.5
Full Structured Issue	**44.7**	**99.2**	**75.5**	**73.1**

Table 5 shows a clear advantage for the full Issue representation across all three benchmarks. Neither an unstructured intermediate Issue nor structure alone provides the same improvement.

6.3 Effectiveness of Patch Validation

We evaluate Patch Validation on candidate patches from ProgSpec-Coding. Each patch is labeled according to whether it resolves the target Issue without breaking the current specification, and we compare these labels with the validator's accept/revise decisions. Detailed labeling criteria and metric definitions are provided in Appendix E.

Table 6: **Effectiveness of Patch Validation on ProgSpec-Coding.**

Metric	Value
Validation Precision	62.5%
Validation Recall	86.2%
Accepted-patch Error Rate	37.5%
Regression Rate	22.5%
Good patches: initial	13.8%
Good patches: revised	69.0%

As shown in Table 6, Patch Validation retains most valid patches, with 86.2% recall and 62.5% precision. The proportion of good patches rises from 13.8% before revision to 69.0% after revision. This suggests that Patch Validation not only filters unreliable patches, but also helps improve patches that initially fail the review.

6.4 Effects of Structural Revision

We examine how structural revision contributes to RepoMAS by selectively disabling updates to the Task Graph and Agent Graph. In all variants, the Task Draft can still incorporate newly discovered requirements.

Table 7: **Effects of Task and Agent Graph revision.** Each variant disables one or both forms of structural revision.

Variant	ProgSpec	HumanEval	HotpotQA
Full RepoMAS	**44.7**	**99.2**	**75.5**
w/o Task Graph Revision	25.8	90.1	55.8
w/o Agent Graph Revision	25.8	88.5	56.9
w/o Structural Revision	30.3	87.8	56.4

Table 7 shows that the full revision mechanism consistently performs best across all three benchmarks. On ProgSpec, disabling either Task Graph or Agent Graph revision reduces the score from 44.7 to 25.8. Disabling both revisions yields 30.3. This result highlights the importance of keeping the task structure and agent organization synchronized after the specification changes. Updating only one graph can leave the execution structure inconsistent, while RepoMAS jointly adapts both structures to the new specification.

6.5 Requirement-Level Composition

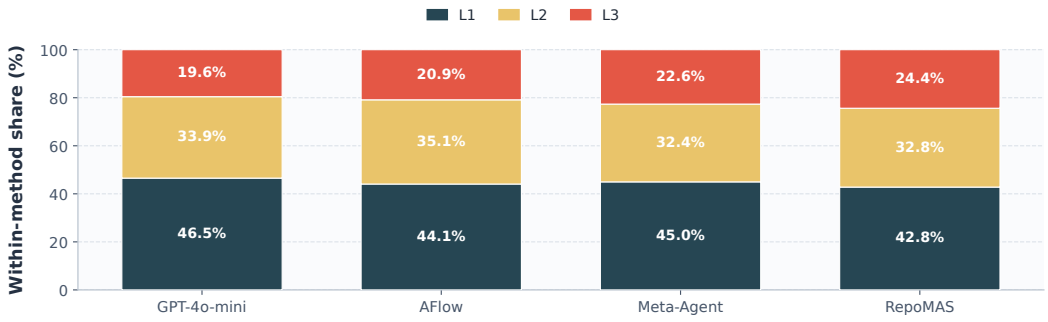


Figure 3: **Composition of satisfied requirements by specification level on ProgSpec.** Each bar shows the within-method distribution of satisfied requirements across explicit (L1), implicit (L2), and latent (L3) requirements.

We further analyze the composition of requirements satisfied by different methods across L1, L2, and L3. Figure 3 reports the within-method distribution, normalized over the requirements satisfied by each method.

The share of L3 requirements increases from 19.6% for the single-agent baseline to 24.4% for RepoMAS, while the share of L1 decreases from 46.5% to 42.8%. L2 remains relatively stable. This suggests that RepoMAS satisfies a larger proportion of latent requirements rather than concentrating mainly on requirements explicit in the initial request.

6.6 Requirement Discovery Continues Throughout Execution

We analyze when RepoMAS first identifies L3 requirements during execution. Figure 4 shows their distribution across *Execution*, *Tool Feedback*, and *Final Verification*, normalized within each domain

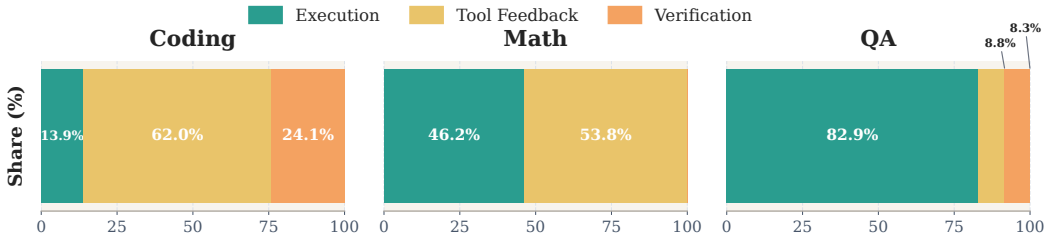


Figure 4: **First discovery stages of L3 requirements identified by RepoMAS with GPT-4o-mini on ProgSpec.** Percentages are normalized over discovered L3 requirements within each domain.

over requirements first identified in these three stages. In Mathematics, Tool Feedback is generated during execution by independently recomputing candidate results, identifying incorrect calculations or derivation steps, and requesting targeted corrections. In contrast, Final Verification checks the completed output before release and identifies requirements that remain missing or unsatisfied after execution.

The discovery pattern varies across domains. In Coding, most discoveries occur through Tool/Verifier Feedback (62.0%), followed by Final Verification (24.1%). In Mathematics, discoveries are split between Execution (46.2%) and verifier feedback (53.8%), with no requirements first identified during Final Verification. In QA, most discoveries occur during Execution (82.9%). These results show that requirement discovery occurs at different stages of the workflow.

6.7 Token Cost Analysis

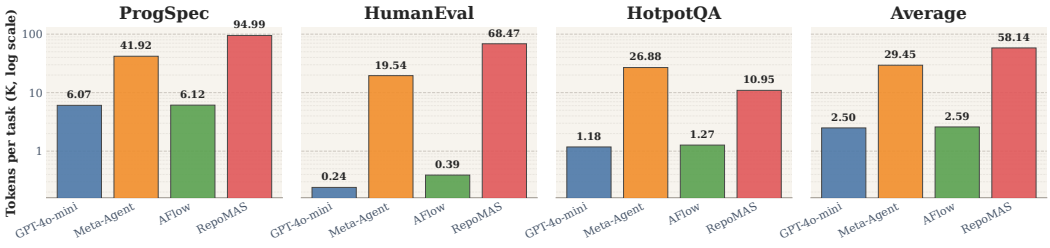


Figure 5: **Token consumption across three benchmarks.** We report the average number of tokens consumed per task (K) on ProgSpec, HumanEval, and HotpotQA using a logarithmic scale.

Figure 5 reports the average token usage per task across the three benchmarks. RepoMAS consumes more tokens than the baselines due to the additional computation introduced by issue tracking, branch-based revision, patch validation, and iterative repair. The largest overhead appears on ProgSpec and HumanEval, where RepoMAS performs more extensive repository updates and validation during execution. On HotpotQA, the gap is substantially smaller. Overall, the results show that the token cost of RepoMAS is closely tied to the amount of repository maintenance required by the task. Benchmarks involving more revisions and validation steps incur higher token consumption.

7 Conclusion

We study *progressively specified tasks*, where task requirements may only become clear during reasoning, tool use, or execution. We introduce **ProgSpec**, a 270-task benchmark across Coding, Mathematics, and document-grounded QA, and propose **RepoMAS**, an issue-driven multi-agent framework inspired by open-source project management. RepoMAS maintains an evolving task specification through structured Issues, patches, validation, and selective re-execution. With GPT-4o-mini, RepoMAS achieves 44.7% weighted requirement coverage on ProgSpec, 1.8 percentage points above the strongest evaluated baseline, and the highest reported scores on five conventional benchmarks. These results motivate systems that revise their explicit representation of task requirements as new evidence becomes available.

AI USE STATEMENT

AI tools were used for language polishing, code assistance, manuscript formatting, and limited editing of technical descriptions. They also supported routine programming and document-preparation tasks. All research decisions, methodological design, dataset construction, implementation choices, experimental analysis, and interpretation of results were carried out by the authors. All AI-assisted outputs, including code and text, were reviewed and verified by the authors before being incorporated into the final manuscript.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje Karlsson, Jie Fu, and Yemin Shi. Autoagents: A framework for automatic agent generation. 2023.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *International Conference on Learning Representations*, volume 2024, pp. 20094–20136, 2024.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Yufan Dang, Chen Qian, Xueheng Luo, Jingru Fan, Zihao Xie, Ruijie Shi, Weize Chen, Cheng Yang, Xiaoyin Che, Ye Tian, et al. Multi-agent collaboration via evolving orchestration. *Advances in neural information processing systems*, 38:165025–165059, 2026.

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. DROP: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In Jill Burstein, Christy Doran, and Thamar Solorio (eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 2368–2378, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1246. URL https://aclanthology.org/N19-1246/.

Song Feng, Siva Sankalp Patel, Hui Wan, and Sachindra Joshi. MultiDoc2Dial: Modeling dialogues grounded in multiple documents. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 6162–6176, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.498. URL https://aclanthology.org/2021.emnlp-main.498/.

Florian Grötschla, Luis Müller, Jan Tönshoff, Mikhail Galkin, and Bryan Perozzi. Agentsnet: Coordination and collaborative reasoning in multi-agent llms. *arXiv preprint arXiv:2507.08616*, 2025.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. 2024.

Grant Hamblin, Kevin Song, Zhanda Zhu, Anand Jayarajan, Sihang Liu, Nandita Vijaykumar, and Gennady Pekhimenko. Specbench: Evaluating specification-level reasoning for software engineering llm agents. *arXiv preprint arXiv:2605.30314*, 2026.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Leng Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, Jie Liu, Lei Qi, Zhiyuan Liu, and Maosong Sun. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems, 2024.

Junda He, Christoph Treude, and David Lo. Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–30, 2025.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *NeurIPS*, 2021.

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

Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.

Dong Li, Yanchi Liu, Xujiang Zhao, Wei Cheng, Zhengzhang Chen, Xintao Wu, Zhong Chen, Chen Zhao, and Haifeng Chen. Coupled hierarchical search over topology and execution for agentic workflow synthesis. *arXiv preprint arXiv:2607.21609*, 2026.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in neural information processing systems*, 36:51991–52008, 2023.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, volume 2024, pp. 52989–53046, 2024.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. A dynamic llm-powered agent network for task-oriented agent collaboration. *arXiv preprint arXiv:2310.02170*, 2023.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *International Conference on Learning Representations*, volume 2024, pp. 9025–9049, 2024.

Elizabeth Mieczkowski, Alexander Ku, Tiwalayo Eisape, Dilip Arumugam, John Matters, Katherine M Collins, Ilia Sucholutsky, and Thomas L Griffiths. Improving the efficiency of language agent teams with adaptive task graphs. *arXiv preprint arXiv:2605.06320*, 2026.

Zhongyuan Peng, Dan Huang, Chuyu Zhang, Caijun Xu, Changyi Xiao, Shibo Hong, David Lo, Lin Qiu, Xuezhi Cao, Jiyuan He, et al. Icae-bench: Evaluating coding agents as interactive project builders. *arXiv preprint arXiv:2607.21217*, 2026.

Chen Qian, Zihao Xie, YiFei Wang, Wei Liu, Kunlun Zhu, Hanchen Xia, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, et al. Scaling large language model-based multi-agent collaboration. In *International Conference on Learning Representations*, volume 2025, pp. 41488–41505, 2025.

Jianhao Ruan, Zhihao Xu, Yiran Peng, Fashen Ren, Zhaoyang Yu, Xinbing Liang, Jinyu Xiang, Yongru Chen, Bang Liu, Chenglin Wu, et al. Aorchestra: Automating sub-agent creation for agentic orchestration. *arXiv preprint arXiv:2602.03786*, 2026.

Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D Nguyen. Multi-agent collaboration mechanisms: A survey of llms. *arXiv preprint arXiv:2501.06322*, 2025.

Siyu Wang, Ruotian Lu, Zhihao Yang, Yuchao Wang, Yanzhou Zhang, Lei Xu, Qimin Xu, Guojun Yin, Cailian Chen, and Xinping Guan. Agentconductor: Topology evolution for multi-agent competition-level code generation. *arXiv preprint arXiv:2602.17100*, 2026a.

Yanbo Wang, Zixiang Xu, Yue Huang, Xiangqi Wang, Zirui Song, Lang Gao, Chenxi Wang, Robert Tang, Yue Zhao, Arman Cohan, et al. Dyflow: Dynamic workflow framework for agentic reasoning. *Advances in Neural Information Processing Systems*, 38:174148–174181, 2026b.

Yimeng Wang, Jiaxing Zhao, Hongbin Xie, Hexing Ma, Yuzhen Lei, Shuangxue Liu, Xuan Song, Zichen Zhang, and Haoran Zhang. Metagen: Self-evolving roles and topologies for multi-agent llm reasoning. *arXiv preprint arXiv:2601.19290*, 2026c.

Yifan Wu, Zhuokai Zhao, Songlin Li, Ho Hin Lee, Jiacheng Zhu, Shirley Wu, Tianhe Yu, Serena Li, Lizhu Zhang, Xiangjun Fan, et al. Swe-together: Evaluating coding agents in interactive user sessions. *arXiv preprint arXiv:2606.29957*, 2026.

Andy Xu and Yu-Wing Tai. Meta-agent: From task descriptions to verified multi-agent systems. *arXiv preprint arXiv:2605.25233*, 2026.

Chen Xu, Yicheng Hu, Ruizi Wang, Xinyu Lin, Wenjie Wang, Dongrui Liu, and Fuli Feng. Tacomas: Test-time co-evolution of topology and capability in llm-based multi-agent systems. *arXiv preprint arXiv:2605.09539*, 2026.

Yingxuan Yang, Chengrui Qu, Muning Wen, Laixi Shi, Ying Wen, Weinan Zhang, Adam Wierman, and Shangding Gu. Understanding agent scaling in llm-based multi-agent systems via diversity. *arXiv preprint arXiv:2602.03794*, 2026.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pp. 2369–2380, 2018.

Junwei Yu, Yepeng Ding, and Hiroyuki Sato. Dyntaskmas: A dynamic task graph-driven framework for asynchronous and parallel llm-based multi-agent systems. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 35, pp. 288–296, 2025.

Guibin Zhang, Yanwei Yue, Xiangguo Sun, Guancheng Wan, Miao Yu, Junfeng Fang, Kun Wang, Tianlong Chen, and Dawei Cheng. G-designer: Architecting multi-agent communication topologies via graph neural networks. *arXiv preprint arXiv:2410.11782*, 2024.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet. *arXiv preprint arXiv:2502.04180*, 2025a.

Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Yu, and Tianlong Chen. Cut the crap: An economical communication pipeline for llm-based multi-agent systems. In *International Conference on Learning Representations*, volume 2025, pp. 75389–75428, 2025b.

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

Yuzhe Zhang, Feiran Liu, Yi Shan, Xinyi Huang, Xin Yang, Yueqi Zhu, Xuxin Cheng, Cao Liu, Ke Zeng, Terry Jingchen Zhang, et al. Silo-bench: A scalable environment for evaluating distributed coordination in multi-agent llm systems. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 29379–29398, 2026.

Kunlun Zhu, Hongyi Du, Zhaochen Hong, Xiaocheng Yang, Shuyi Guo, Daisy Zhe Wang, Zhenhailong Wang, Cheng Qian, Xiangru Tang, Heng Ji, et al. Multiagentbench: Evaluating the collaboration and competition of llm agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8580–8622, 2025.

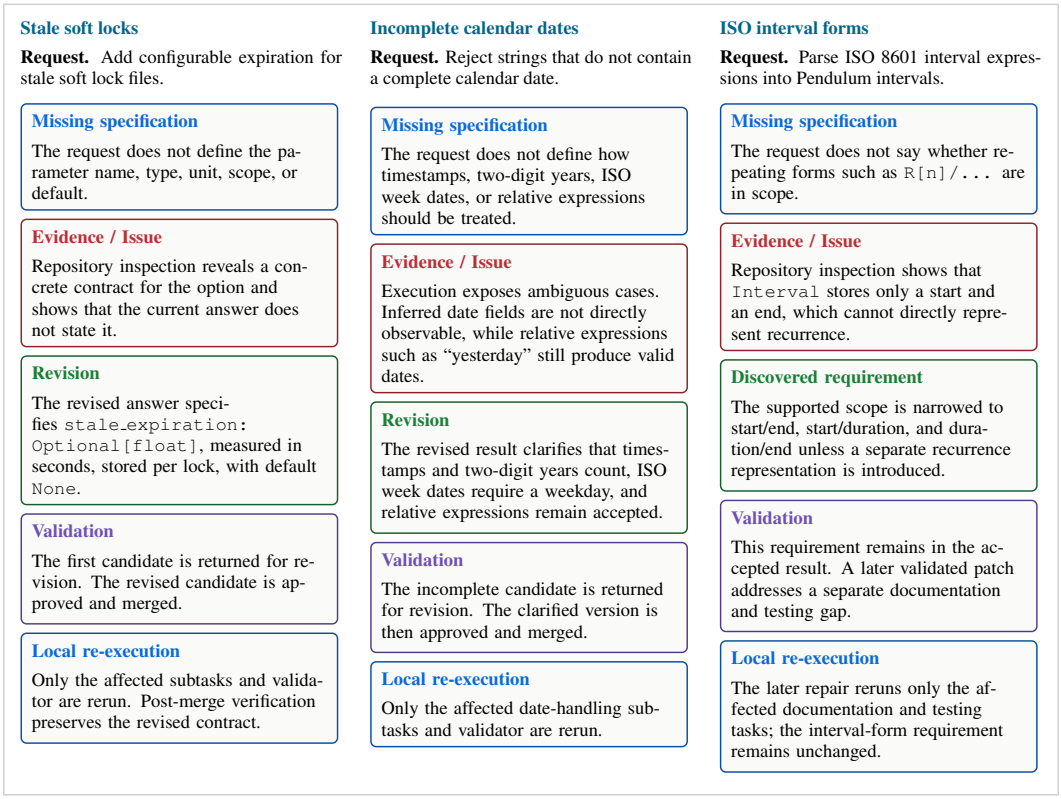


Figure 6: **Examples of requirement discovery in ProgSpec-Coding.** Execution and repository evidence expose requirements that are absent from the initial requests. In the first two cases, the discovered requirements trigger revision, validation, and local re-execution. In the Pendulum case, the interval-scope requirement is discovered and retained, while the later validated repair addresses a separate gap.

A Case Study

Figure 6 shows three ProgSpec-Coding examples in which execution reveals requirements that are not stated in the initial request. In the first two cases, RepoMAS records the missing requirement as an Issue, revises the affected result, and locally re-executes the relevant part of the workflow after validation. The third case illustrates requirement discovery from repository structure, while the later accepted repair addresses a separate documentation and testing gap.

The cases illustrate that specification discovery can arise from different forms of execution evidence. RepoMAS makes these missing requirements explicit and incorporates them into subsequent reasoning without restarting the full workflow.

B ProgSpec Construction Details

This section describes the construction of ProgSpec. ProgSpec contains Coding, Mathematics, and document-grounded QA tasks. Each gold requirement must be supported by the task context or be discoverable during execution. Three graduate students independently annotated all 270 ProgSpec tasks. Each annotator reviewed the full set and assigned requirement-level judgments. We then compared the three annotations for every task. Cases with inconsistent labels were discussed and re-examined against the task description and supporting evidence. The final annotation was determined after resolving these disagreements, yielding a single consolidated set of labels for the benchmark.

B.1 Coding Task Construction

The Coding split contains repository-grounded implementation-planning tasks built from public Python projects on GitHub. Given a change request and a fixed repository snapshot, the system must produce an implementation plan that captures the behavior required by the task and identifies the relevant parts of the codebase. We evaluate the plan against atomic requirements rather than by executing a code patch.

We select repositories that contain multiple interacting files or modules and require reasoning across component boundaries. Each repository must also provide development records, such as issues, pull requests, commits, or tests, that can be used to reconstruct task requirements. We exclude repositories whose execution depends on unavailable external services, complex environment configurations, or insufficient development evidence. For each repository, we restrict the task scope to the files and modules directly involved in the target change, while retaining their relevant dependency relations.

The 12 repositories used in ProgSpec are:

> *attrs*, *cattrs*, *dateparser*, *filelock*, *pendulum*, *pluggy*, *pydantic-settings*, *python-dotenv*, *requests-cache*, *tenacity*, *tomlkit*, and *typer*.

Each task is created from a natural-language change request and a fixed repository snapshot. For requirements that are not explicit in the request, we record the evidence used during annotation and separately track whether that evidence is available to the solving system. Annotation may draw on repository history, while inference-time evidence comes from the released codebase, available tests, and observable execution feedback.

We retain only requirements that are necessary for the requested change and recoverable from the allowed task context. Historical preferences or optional design choices are excluded when the current repository does not provide enough evidence to make them mandatory.

B.2 Mathematics Task Construction

The Mathematics split contains 60 problems from MATH (Hendrycks et al., 2021) and 30 problems from the text-only subset of OlympiadBench (He et al., 2024).

We select problems with well-defined final answers and enough internal structure to support multiple independently verifiable requirements. Problems that depend on images, external tools, or ambiguous assumptions are excluded.

For each problem, we inspect both the problem statement and its reference solution. We derive candidate requirements that are necessary for a complete and correct answer without treating the reference derivation as the only valid solution path. The resulting requirements may include the requested target, domain restrictions, assumptions, intermediate conditions, boundary cases, or checks needed to ensure that the final result is valid.

We avoid treating individual derivation steps as separate requirements. Direct implications of the statement and standard mathematical domain conditions are labeled L2. L3 requirements concern task-specific conditions whose necessity is established through further analysis or verification, rather than through the direct application of a standard domain rule. Such conditions must constrain the validity or completeness of the solution independently of the chosen derivation. Requirements tied only to one reference solution path are excluded, and alternative correct derivations are accepted.

B.3 QA Task Construction

The QA split contains 90 document-grounded tasks constructed from MultiDoc2Dial (Feng et al., 2021). Each task provides an initial user request together with the documents needed to answer it.

We select tasks where a complete answer requires combining information from multiple parts of the documents rather than retrieving a single sentence. Some requirements are not stated directly in the user request, but can be recovered from the provided document context.

Only the initial user request and document context are given to the system. Later dialogue turns are not used as task input or as sources of gold requirements. This ensures that latent requirements come from the available documents rather than from later user clarification.

For each gold requirement, we record the supporting document passage. Requirements without support in the provided documents are excluded.

C ProgSpec Annotation Guidelines

C.1 Atomic Requirement Annotation

For each task, annotate a set of atomic requirements that together define what a successful solution must satisfy. These requirements form the reference set $\mathcal{R}^*$.

Each requirement must satisfy the following rules:

- **Atomicity:** Write one requirement per item. Split independent conditions into separate requirements.
- **Verifiability:** Phrase the requirement so that its satisfaction can be checked from the final output or execution result.
- **Grounding:** Include a requirement only when it is supported by the task statement or the evidence available for that task.

Annotate requirements that are necessary for successful task completion. These may cover expected behavior, boundary conditions, reasoning constraints, or facts that must appear in the final answer.

Do not annotate optional improvements, stylistic preferences, or implementation choices that are not required by the task.

C.2 Requirement Levels

Assign each requirement to one of the following levels according to its supporting evidence and derivation:

1. **L1 — Explicit**: The requirement is explicitly stated in the initial user request.
2. **L2 — Implicit**: The requirement is a direct consequence of the request or a standard domain condition, such as the domain restriction of a stated mathematical expression.
3. **L3 — Latent**: Establishing the requirement requires locating and connecting task-specific evidence, or carrying out task-specific analysis or verification. Relevant evidence may come from supplied repository files, document passages, tool outputs, or intermediate reasoning results.

Classify directly stated requirements as L1 and direct implications or standard domain conditions as L2, even if a system discovers them later. For each L3 requirement, record the specific evidence or derivation that establishes it.

C.3 Annotation Procedure

Each task is annotated using the following procedure:

1. Inspect the initial task request and the available supporting evidence.
2. Identify the conditions that must hold for the task to be considered successfully completed.
3. Decompose these conditions into atomic and independently verifiable requirements.
4. Assign each requirement an L1, L2, or L3 label according to the evidence criteria in the requirement-level definitions.
5. Record both annotation evidence and inference-accessible evidence, together with the action needed to acquire the latter.
6. Review the final requirement set for missing, duplicated, or overly broad requirements.

C.4 Annotation Examples

Table 8 shows representative examples of the three requirement levels.

Table 8: **Examples of ProgSpec requirement levels.**

Level	Example and evidence
L1 (Explicit)	The user asks the function to return a JSON object. This requirement is stated directly in the request.
L2 (Implicit)	The task asks for the real-valued expression $\log(x-1)$ to be evaluated. The condition $x > 1$ follows directly from the domain of the real logarithm, although it is not stated in the request.
L3 (Latent)	The request does not specify how empty input should be handled. Inspection of an available repository test shows that empty input must return an empty result rather than raise an exception.

C.5 Quality Control

After the initial annotation, each task should be reviewed by another annotator. The reviewer checks the following aspects:

- **Validity:** Every requirement must be necessary for successful task completion and supported by available evidence.
- **Atomicity:** Each item should express one independently verifiable requirement without unnecessary overlap with other items.
- **Evidence:** The recorded evidence must be sufficient to support the annotated requirement.
- **Level:** The L1–L3 label must follow the requirement-level definitions in the annotation guidelines.

Requirements that fail any of these checks are revised or removed before the final annotation is accepted.

D ProgSpec Evaluation Details

D.1 Weighted Requirement Coverage

ProgSpec evaluates the final artifact against the atomic requirements of each task. Requirements are weighted by their annotation level:

$$w_i = \begin{cases} 1, & r_i \in \mathcal{R}_{\mathrm{L1}}, \\ 2, & r_i \in \mathcal{R}_{\mathrm{L2}}, \\ 3, & r_i \in \mathcal{R}_{\mathrm{L3}}. \end{cases} \tag{7}$$

Let $\mathcal{S}_q \subseteq \mathcal{R}_q^*$ denote the requirements satisfied by the final artifact for task q. Its weighted requirement coverage is

$$\mathrm{Coverage}(q) = 100 \times \frac{\sum_{r_i \in \mathcal{S}_q} w_i}{\sum_{r_i \in \mathcal{R}_q^*} w_i}. \tag{8}$$

For a collection of tasks $\mathcal{Q}$, we pool requirement weights across tasks:

$$\mathrm{Coverage}_{\mathcal{Q}} = 100 \times \frac{\sum_{q \in \mathcal{Q}} \sum_{r_i \in \mathcal{S}_q} w_i}{\sum_{q \in \mathcal{Q}} \sum_{r_i \in \mathcal{R}_q^*} w_i}. \tag{9}$$

The weighting gives greater emphasis to requirements that are not explicitly stated in the initial request. Reported domain and overall scores are computed from the pooled requirement weights rather than by averaging task scores.

D.2 Requirement Satisfaction

Each atomic requirement is evaluated against the system's final artifact. The evaluator uses the requirement and its supporting evidence to determine whether the required condition is satisfied.

For Coding, evaluation is performed on the final implementation plan. For Mathematics, equivalent correct derivations and answers are accepted. For QA, the final answer must contain the information supported by the supplied documents. A requirement is marked as satisfied only when the final artifact meets it without contradicting the available evidence.

E Patch Validation Evaluation

We evaluate Patch Validation on candidate patches collected from ProgSpec-Coding. The online validator decides whether each candidate patch is accepted or returned for revision. We compare this decision with an offline quality label.

Patch quality. A patch is labeled *good* when it resolves the target Issue and produces a revision consistent with the task requirements and available evidence. Previously established requirements must remain satisfied unless new evidence shows that they should be revised. All other patches are labeled *bad*. The offline evaluator has access to the annotated requirements, while the online validator does not.

Validation metrics. Let N_{AG} and N_{AB} denote accepted good and accepted bad patches, and let N_{RG} denote rejected good patches. We compute

$$\mathrm{Precision} = \frac{N_{\mathrm{AG}}}{N_{\mathrm{AG}} + N_{\mathrm{AB}}}, \qquad \mathrm{Recall} = \frac{N_{\mathrm{AG}}}{N_{\mathrm{AG}} + N_{\mathrm{RG}}}. \tag{10}$$

The Accepted-patch Error Rate is the fraction of accepted patches labeled bad, and is therefore $1 - \mathrm{Precision}$.

We also measure whether an accepted patch breaks requirements that were valid before revision:

$$\mathrm{Regression\ Rate} = \frac{N_{\mathrm{AR}}}{N_{\mathrm{A}}}, \tag{11}$$

where N_{AR} is the number of accepted patches that introduce a regression and N_{A} is the total number of accepted patches. All metrics are reported as percentages.

F Backbone Robustness

To examine whether the performance of RepoMAS depends on a specific backbone model, we evaluate three additional backbones, which are GPT-5.5, Gemini 3.7 Flash, and DeepSeek-V4-Flash. Table 9 reports the results.

Within each backbone group, we use the same benchmark splits and evaluation metrics as in the main experiment. Except for the backbone model, the evaluated configurations use the same prompts and benchmark settings.

Results across additional backbones further demonstrate the applicability of RepoMAS beyond GPT-4o-mini, with improvements observed across most of the model families and tasks.

Table 9: **Backbone robustness results on six benchmarks.** We repeat the main experiments using different backbone models while keeping the methods and evaluation settings unchanged. All methods within each group use the same backbone model. Bold marks the highest score in each column.

Backbone	Method	HumanEval	HotpotQA	DROP	GSM8K	MBPP	ProgSpec
GPT-5.5	GPT-5.5	99.24	59.38	39.12	93.46	83.28	92.08
	MetaGPT (Hong et al., 2024)	98.47	32.88	3.38	71.18	81.23	89.95
	AgentCoder (Huang et al., 2023)	99.24	50.62	33.75	92.61	83.58	93.02
	G-Designer (Zhang et al., 2024)	**100.00**	57.00	36.50	93.36	84.46	94.20
	AFlow (Zhang et al., 2025c)	99.24	57.75	39.12	**94.22**	83.58	93.34
	Meta-Agent (Xu & Tai, 2026)	98.47	54.37	25.50	92.89	80.06	89.50
	TacoMAS (Xu et al., 2026)	99.24	56.62	34.38	92.70	81.23	92.55
	LATTE (Mieczkowski et al., 2026)	99.24	46.00	28.62	92.32	82.11	92.06
	RepoMAS	**100.00**	**70.00**	**66.89**	82.84	**94.13**	**94.45**
Gemini 3.7 Flash	Gemini 3.7 Flash	**100.00**	65.00	49.50	95.45	89.44	67.48
	MetaGPT (Hong et al., 2024)	97.71	44.38	25.25	81.04	71.26	92.08
	AgentCoder (Huang et al., 2023)	**100.00**	63.62	45.00	61.04	**90.91**	90.85
	G-Designer (Zhang et al., 2024)	**100.00**	65.88	47.38	78.01	89.44	92.71
	AFlow (Zhang et al., 2025c)	**100.00**	65.50	48.75	94.50	88.56	89.95
	Meta-Agent (Xu & Tai, 2026)	**100.00**	63.50	40.62	93.55	87.10	90.14
	TacoMAS (Xu et al., 2026)	**100.00**	65.50	47.75	95.64	89.15	89.41
	LATTE (Mieczkowski et al., 2026)	**100.00**	61.50	47.38	67.11	89.15	**94.46**
	RepoMAS	**100.00**	**72.63**	**67.25**	**97.44**	90.62	94.13
DeepSeek-V4-Flash	DeepSeek-V4-Flash	99.24	**58.13**	40.62	93.27	82.40	70.63
	MetaGPT (Hong et al., 2024)	99.24	38.38	17.38	80.09	80.94	66.37
	AgentCoder (Huang et al., 2023)	97.71	55.38	36.62	92.99	82.70	67.45
	G-Designer (Zhang et al., 2024)	96.95	54.62	37.12	91.75	81.82	67.86
	AFlow (Zhang et al., 2025c)	99.24	55.88	41.50	92.61	80.94	63.93
	Meta-Agent (Xu & Tai, 2026)	**100.00**	49.88	28.62	89.95	81.23	68.59
	TacoMAS (Xu et al., 2026)	99.24	54.00	35.12	91.28	**82.99**	65.80
	LATTE (Mieczkowski et al., 2026)	99.24	46.12	22.75	83.70	81.52	67.72
	RepoMAS	**100.00**	57.75	**42.37**	**94.98**	80.94	**73.72**

G Annotation Quality and Human Validation

We conduct an independent human review of a fixed ProgSpec annotation snapshot. The review covers 90 randomly sampled tasks and 500 requirements. Five graduate students who were not involved in the original annotation conduct the review.

Review protocol. At the requirement level, reviewers first determine whether each requirement is necessary for completing the task. They then check whether it is independently testable, supported by the available evidence, and stated unambiguously. These judgments are used to measure requirement validity, independence, evidence sufficiency, and ambiguity.

At the task level, reviewers examine the annotation as a whole. They check whether the requirement set is supported by the task context, sufficiently complete, properly decomposed into atomic requirements, and assigned to the appropriate L1–L3 levels. We compare these judgments with the original annotations to measure annotation–review agreement.

Results. Table 10 summarizes the human validation results. Among the 500 reviewed requirements, 477 (95.4%) are judged valid, 491 (98.2%) satisfy the independence criterion, and 487 (97.4%) have sufficient supporting evidence. Only one requirement is judged ambiguous.

Across the four task-level judgments for 90 tasks, the original annotations agree with the independent review in 329 of 360 cases, corresponding to 91.4% agreement and a Gwet's AC1 of 0.91. Agreement is highest for contextual support at 98.9%, followed by completeness and atomicity at 90.0%, and requirement-level assignment at 86.7%.

Overall, the review shows that ProgSpec requirements are consistently grounded in task evidence and can be evaluated as independent task conditions.

Table 10: **Human validation of ProgSpec annotations.** Results are computed over 90 tasks and 500 annotated requirements.

Metric	Score	95% CI	Evaluation Unit
Annotation–review agreement	91.4% (AC1 = 0.91)	[88.0, 93.9]	329 / 360 judgments
Requirement validity	95.4%	[93.2, 96.9]	477 / 500 requirements
Requirement independence	98.2%	[96.6, 99.1]	491 / 500 requirements
Evidence sufficiency	97.4%	[95.6, 98.5]	487 / 500 requirements
Ambiguity rate	0.2%	[0.0, 1.1]	1 / 500 requirements

H Experiment Configuration

This section describes the common experimental settings used for RepoMAS. Unless otherwise specified, all experiments use the full RepoMAS framework, including structured Issues, patch validation, structural revision, and local re-execution.

H.1 Execution Environment

Experiments are run on Linux with Python 3.11. Model inference is served through an OpenAI-compatible API, while the RepoMAS runtime itself does not require GPU computation. Structured model outputs are constrained by JSON schemas to ensure that intermediate artifacts can be parsed and reused by subsequent stages. All benchmark inputs are fixed before inference. ProgSpec-Coding additionally provides a repository snapshot for each task.

H.2 Runtime Settings

Unless otherwise specified, RepoMAS uses at most three execution–revision iterations per task and one task attempt. Model outputs are limited to 4,096 tokens per model call. ProgSpec runs use temperature 0, while other benchmark runs use temperature 0.2. The evaluation settings for each benchmark are summarized in Table 11.

Table 11: **Evaluation settings across benchmarks.** We report the temperature, maximum number of generated tokens, and maximum number of iterations used for each benchmark.

Benchmark	Temperature	Max Tokens	Max Iterations
HumanEval	0.2	4096	3
HotpotQA / DROP / GSM8K / MBPP	0.2	4096	3
ProgSpec–Coding	0.0	4096	3
ProgSpec–Math / QA	0.0	4096	3

For the additional-backbone experiments, we keep the RepoMAS configuration fixed wherever possible and change only the backbone model. These runs use the same full Issue representation, patch validation, structural revision, and local re-execution procedure.

H.3 Prompt Protocol

All model calls use structured prompts with explicit output schemas. Each call receives the original task together with the information required by its current Task Graph node. Depending on the stage, this may include upstream artifacts, open Issues, or the current candidate under revision. Gold requirements, hidden tests, and reference answers are never included in generation prompts.

The initialization stage asks the model to produce a structured Task Draft from the public request. RepoMAS then compiles this draft into the Task Graph and Agent Graph. Worker agents execute individual subtasks and return structured artifacts. When execution reveals a problem, the corresponding evidence is recorded as an Issue and used to construct a candidate patch.

The final synthesizer receives the accepted artifacts produced during execution and generates the final response to the original task. Patch Validation is performed independently of the contributor

that proposed the revision. An accepted patch updates the repository state, after which only the affected part of the workflow is re-executed.

H.4 Evaluator Protocol

After inference, ProgSpec is evaluated by a separate post-hoc LLM judge using `GPT-5.5`. For each task, the judge receives the public task input, evaluator-only atomic requirements, the reference answer when available, and the system response. It returns a structured verdict for each requirement, including a binary coverage decision and supporting evidence from the response.

H.5 Statistical Significance

To systematically assess the statistical significance of RepoMAS's improvements, we conduct hypothesis tests across all datasets and all baseline methods. Under the same evaluation setting, each experiment is repeated 10 times, with RepoMAS and the corresponding baseline evaluated on the same task set in each run, yielding paired performance measurements. For each dataset–baseline pair, we apply a two-sided paired t-test to the 10 paired results.

For each comparison, the null hypothesis assumes no difference in mean performance between RepoMAS and the corresponding baseline. We use a significance level of $\alpha = 0.05$ and report the resulting p-values across all dataset–baseline combinations. This analysis examines whether the observed performance gains of RepoMAS are consistently supported by statistical evidence rather than being attributable to run-to-run variation.