RESPEC: LEARNING PAPER REPRODUCTION VIA SYNTHETIC SUPERVISION AND PROCESS REWARDS

Anonymous authors
Paper under double-blind review

ABSTRACT

Faithful paper-to-code reproduction requires translating a paper's technical specifications into an implementation. Learning this capability requires intermediate supervision and fidelity feedback, which paper-repository pairs alone do not provide. We present ReSpec, a framework that automatically constructs synthetic supervision and checklist-based process rewards from papers. Its data engine selects 6,172 paper-repository pairs through implementation-fidelity assessment and synthesizes 430,428 examples for six reproduction sub-tasks. It also constructs paper-grounded checklists to define fine-grained rewards for the plan pipeline and file-scoped code writing, yielding 76,448 samples for parallel training through task-decomposed reinforcement learning. Using these paper-derived signals, we train the ReSpec model through supervised fine-tuning followed by reinforcement learning. On PaperBench Code-Dev, the ReSpec model equipped with the DeepCode-Base scaffold and two-pass plan refinement outperforms Qwen3.5-35B-A3B in the DeepCode-Base scaffold by 5.2 points (70.6 vs. 65.4) with one-sixth as many active parameters (0.5B), while approaching Claude-Sonnet-4.5 (73.5) and GLM-4.7 (75.0). These results demonstrate that automatically constructed process supervision and rewards can turn paper-to-code reproduction into a learned capability in a compact specialist.

1 INTRODUCTION

Reproducibility makes scientific findings verifiable and reusable (Pineau et al., 2021). LLM agents now combine reasoning with tool use (Yao et al., 2023) and navigate, edit, and test code repositories (Yang et al., 2024; Jimenez et al., 2024). Structured planning and code retrieval further support repository generation and reuse (Luo et al., 2026a; Wang et al., 2025). Reproducing a machine learning paper adds a specification challenge: requirements distributed across equations, figures, and experimental protocols must be translated into a implementation (Seo et al., 2026; Lin et al., 2025b). Executable code can still omit an algorithmic step or misimplement a mathematical expression (Zhou et al., 2025; Li et al., 2025). PaperBench makes this distinction explicit by evaluating reproductions against human-authored rubrics (Starace et al., 2025). We study implementation fidelity at this granularity: whether the generated code satisfies the paper's implementation requirements.

Existing paper-to-code systems address this challenge through complementary inference-time mechanisms. One line of work organizes the conversion from paper content to code through structured plans and staged generation (Seo et al., 2026; Lin et al., 2025b; Li et al., 2025). Another supplements underspecified implementation details with knowledge retrieved from related literature and code (Zhao et al., 2026; Luo et al., 2026b; Li et al., 2025). Other approaches use verification feedback to revise intermediate outputs or generated code, improving adherence to paper requirements and execution correctness (Zhou et al., 2025; Lin et al., 2025a; Li et al., 2025). These approaches organize context, tools, and revision procedures to improve reproduction attempts. In our work, we aim to internalize paper-reproduction ability into the model by constructing dedicated training data.

However, available papers and repositories do not directly provide the process supervision and fidelity feedback needed to develop this capability. First, a paper describes what to implement and its paired repository provides a final implementation, while the intermediate analysis, planning, and file-by-file construction process remains unrecorded. Second, full-repository rollouts are costly to collect, while sparse repository-level rewards provide limited guidance for optimizing the plan

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.

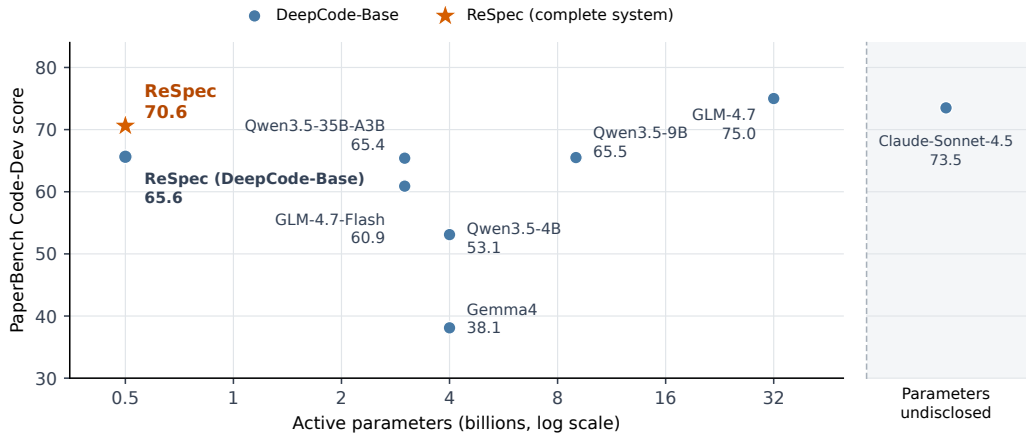

Figure 1. PaperBench Code-Dev score versus active parameter count (log scale). Circles use the DeepCode-Base scaffold; the star shows the complete ReSpec system with two-pass refinement.

pipeline and file-scoped code writing. Further, manually authored rubrics cannot scale to support training across many papers with task-specific criteria for plans and individual code files. These gaps motivate a central question:
How to construct scalable process supervision and fidelity feedback from scientific papers?

We present **ReSpec** (**Re**production **Spec**ialist), a framework that converts papers into synthetic process supervision and checklist-based process rewards. Its data engine selects reproducible papers by assessing how faithfully their paired repositories implement the stated requirements. From these papers, it synthesizes intermediate supervision for six sub-tasks: paper analysis, plan generation, plan refinement, file ordering, code writing, and memory. This yields 430,428 supervised fine-tuning (SFT) examples from 6,172 selected paper-repository pairs; reference repositories are used only for curation. For feedback during reproduction, the framework automatically constructs checklists of atomic implementation requirements for plans and individual code files. A generative reward model (GRM) evaluates their satisfaction, providing process rewards for reinforcement learning (RL) with GRPO (Shao et al., 2024). The RL corpus contains 76,448 checklist-equipped samples. We use these training signals for SFT followed by RL of a Sigma MoE backbone with 20B total and 0.5B active parameters (Hu et al., 2025), yielding the **ReSpec model**. The *complete ReSpec system* combines this model with the DeepCode-Base scaffold and two-pass plan refinement.

We evaluate the complete ReSpec system on PaperBench and the 90-paper Paper2CodeBench suite. The complete system scores 70.6 on PaperBench, outperforming Qwen3.5-35B-A3B in the DeepCode-Base scaffold by 5.2 points with one-sixth as many active parameters and approaching Claude-Sonnet-4.5 and GLM-4.7 (Figure 1). On Paper2CodeBench, it scores 68.2 under our criterion-level evaluation, exceeding the same baseline by 4.7 points. These results demonstrate the value of automatically constructed supervision and process rewards for learning paper-to-code reproduction.

Our contributions are as follows:

- We develop an automated data engine that collects highly reproducible papers and synthesizes supervision for six reproduction sub-tasks, yielding 430,428 SFT examples. We release the engine and accompanying training data.
- We introduce task-decomposed RL for paper reproduction, combining parallel training of the plan pipeline and file-scoped code writing with checklist-based process rewards.
- We validate these signals by training the ReSpec model. With 0.5B active parameters, the complete system outperforms a model with six times as many active parameters, approaching Claude-Sonnet-4.5 and GLM-4.7 on PaperBench Code-Dev.

2 Related Work

Paper-to-code reproduction. Benchmarks evaluate paper-to-code reproduction at different implementation scopes. PaperBench and Paper2CodeBench assess full-repository reproduction (Starace et al., 2025; Seo et al., 2026), whereas SciReplicate-Bench targets algorithms from NLP papers, ResearchCodeBench targets novel ML contributions, and LMR-BENCH asks agents to reconstruct masked functions in supplied repositories (Xiang et al., 2025; Hua et al., 2025; Yan et al., 2025). Systems address these tasks through planning, knowledge retrieval, and revision. PaperCoder and AutoP2C organize paper content into plans and staged code generation (Seo et al., 2026; Lin et al., 2025b); structured planning also supports general codebase generation in ZeroRepo through repository planning graphs (Luo et al., 2026a). To recover implementation knowledge, AutoReproduce and xKG retrieve related literature and code (Zhao et al., 2026; Luo et al., 2026b). For reuse of existing repositories, RepoMaster organizes code through code and dependency graphs (Wang et al., 2025). Verification then guides revision: RePro uses atomic criteria for targeted code refinement (Zhou et al., 2025), and Lin et al. (2025a) verify stage outputs against system instructions through prompt-free collaboration. DeepCode integrates planning, retrieval, and correction through blueprint distillation, code memory, reference-code retrieval, and execution feedback (Li et al., 2025). These approaches primarily improve inference-time orchestration. The ReSpec framework pursues a complementary training route, turning papers into supervision and process rewards that develop the underlying model's reproduction capability.

Synthetic supervision for coding agents. Synthetic supervision trains models on instructions and interaction trajectories generated from existing resources. Self-Instruct generates and filters instruction examples (Wang et al., 2023), and Fu et al. (2023) distill generated reasoning traces into smaller models. For code generation, Magicoder's OSS-Instruct grounds synthetic coding instructions in open-source snippets (Wei et al., 2024). Extending supervision to tool use, CodeActInstruct provides multi-turn trajectories with executable code actions (Wang et al., 2024b). Collecting such trajectories for repository-level tasks also requires executable environments: SWE-Gym supplies tasks and environments for training agents and trajectory verifiers (Pan et al., 2025), while SWE-smith scales task construction by modifying repositories to break existing tests (Yang et al., 2025). ReSpec applies this data-construction perspective to paper-to-code reproduction. Its data engine uses paired repositories to select reproducible papers, then synthesizes intermediate supervision for six reproduction sub-tasks from the papers' implementation requirements.

Process rewards and rubric-based learning. Beyond demonstrations, feedback can support both inference-time revision and learning. Self-Refine and Reflexion use feedback to revise outputs or guide subsequent attempts (Madaan et al., 2023; Shinn et al., 2023). Process supervision makes feedback available within a solution: Lightman et al. (2024) train mathematical reasoning verifiers with human step-level labels, while Math-Shepherd automatically constructs process supervision for step-level verification and reinforcement learning (Wang et al., 2024a). For open-ended generation, criterion-based rewards specify what such feedback should measure. RLVRR decomposes reference-based rewards into content and style (Jiang et al., 2026), Rubrics as Rewards uses instance-specific criteria for on-policy RL (Gunjal et al., 2026), and OpenRubrics generates rubrics from preference pairs for reward modeling (Liu et al., 2025). ReSpec connects criterion-based evaluation to process rewards for long-horizon reproduction: paper-derived checklists and criterion-level GRM judgments provide feedback for the plan pipeline and file-scoped code writing.

3 Method

ReSpec converts papers into intermediate supervision and checklist-based process rewards through three stages (Figure 2): paper selection, SFT data synthesis, and task-decomposed RL. Paired repositories provide implementation evidence for selection only; they are excluded from supervision and reward construction. A shared backbone is fine-tuned on six reproduction sub-tasks, then jointly optimized with GRPO for the plan pipeline and file-scoped code writing using checklist-based process rewards scored by a generative reward model (GRM). The *plan pipeline* comprises concept and algorithm analysis, initial plan generation, and two-pass refinement.

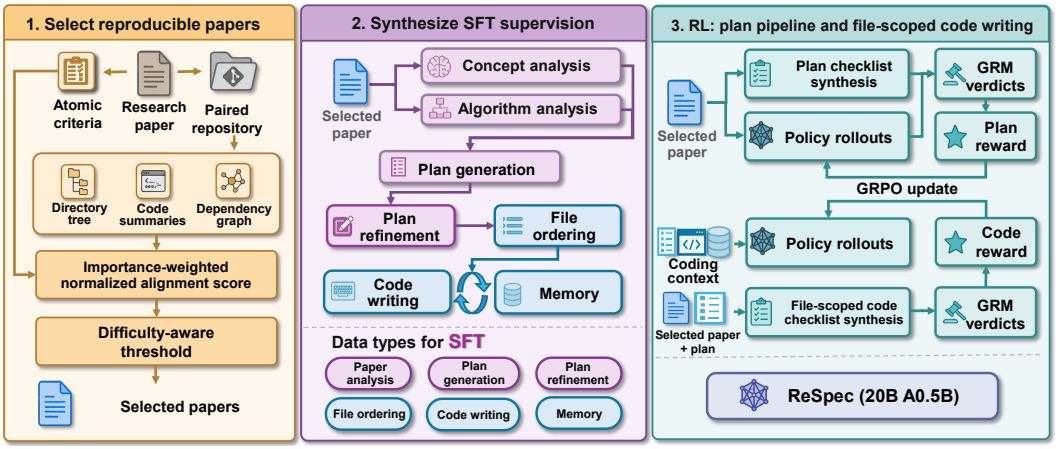

Figure 2. Overview of the ReSpec framework. (1) Paper selection combines importance-weighted repository alignment with difficulty-aware thresholds. (2) Selected papers provide synthetic supervision for six reproduction sub-tasks to fine-tune a backbone. (3) A GRM evaluates plans and code files against paper-grounded checklists, guiding GRPO for the plan pipeline and file-scoped code writing. Together, these stages produce ReSpec with 20B total and 0.5B active parameters.

Weight	Definition
4	**Paper novelty:** a newly proposed loss, algorithmic component, architecture, or pipeline.
3	**Structural logic:** model structure, mathematical formulation, or I/O specification.
2	**Training configuration:** optimizer, learning rate, batch size, epochs, or regularization.
1	**Reproducibility detail:** data preparation, random seeds, environment, or dependencies.
0	**Non-implementation content:** motivation, future work, reported results, or background theory.

Table 1. Importance weights for atomic reproduction criteria. Weight-0 criteria are discarded.

3.1 Selecting Reproducible Papers

We select source papers by measuring how faithfully their paired repositories implement the stated requirements.

Collecting paper-repository pairs. We collect papers from 11 ML and AI venues (2020-2025; Appendix A.1) and associate them with repositories through the Papers with Code dataset[1] When no association is available, we extract GitHub links from the paper and ask an LLM to verify that a candidate repository is the paper's implementation. Candidate repositories must contain 3-100 code files. We exclude all PaperBench and Paper2CodeBench papers to prevent training-test leakage.

Extracting atomic reproduction criteria. Following RePro's atomic verification criteria (Zhou et al., 2025), we use Qwen3-235B-A22B to extract implementation requirements from technical sections, excluding non-technical narrative and background theory. We rewrite each requirement as an independently checkable fact with its applicable scope, such as dataset and split, model variant, or run configuration. Deduplication removes redundant items under identical scopes, with a capped removal budget that preserves distinct values and algorithmic stages.

Assigning importance weights. To prioritize fidelity to the paper's technical contributions, Qwen3-235B-A22B assigns weights $w(c) \in \{0, \ldots, 4\}$ using the manually specified levels in Table 1. The abstract and introduction inform importance but supply no criteria; weight-0 items are discarded. Appendix F.1.1 gives an example of the weighted criteria.

Scoring reproduction quality with the criteria. For each criterion, Qwen3-32B ranks at most k candidate files ($k = 5$) using the repository's directory tree, summaries of files and their classes

[1]Official data repository: `https://github.com/paperswithcode/paperswithcode-data`.

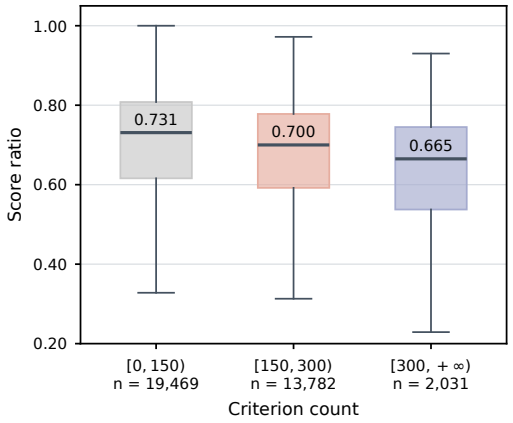

Figure 3. Alignment scores A_{repo} before filtering, grouped by criterion count; n is the candidate-pair count. Lower scores appear in larger-count groups.

Metric	Samples	Tokens (M)
Selected paper pool	6,172	-
Venue / year coverage	11 venues, 2020-2025	
Samples by sub-task		
Paper analysis	11,868	568.0
Plan generation	6,309	278.2
Plan refinement	37,103	1,536.5
File ordering	6,000	136.4
Code writing	209,242	10,095.3
Memory generation	159,906	4,382.6
Total	**430,428**	**16,997.1**

Table 2. Selected paper pool and SFT corpus. Analysis covers concepts and algorithms; refinement includes both passes. Code and memory samples are file-scoped.

and functions, and import dependency graph. It then reads these files together with the criterion to judge whether the code implements it. The n binary judgments are aggregated into an importance-weighted normalized alignment score:

$$A_{\text{repo}} = \frac{\sum_{i=1}^{n} w_i \, x_i}{\sum_{i=1}^{n} w_i} \in [0, 1], \tag{1}$$

where $w_i \in \{1, \ldots, 4\}$ is the criterion's importance weight and $x_i \in \{0, 1\}$ indicates whether it is implemented in the repository.

Difficulty-aware filtering. A single threshold can disproportionately reject papers with more implementation requirements. Since alignment scores tend to decrease with criterion count (Figure 3), we use criterion count as a proxy for reproduction difficulty and apply progressively lower thresholds to higher-count groups (Appendix A.2). Retaining pairs that meet their group's threshold yields 6,172 paper-repository pairs (Table 2).

3.2 Paper-Grounded SFT Data Synthesis

The *ReSpec scaffold* extends DeepCode's paper analysis, plan generation, code writing, and code memory (CodeMem) (Li et al., 2025) with explicit file ordering and two-pass plan refinement. The variant without refinement is the *DeepCode-Base scaffold*. We run Qwen3.5-397B-A17B within the ReSpec scaffold on selected papers and convert its intermediate outputs and tool interactions into supervision for six sub-tasks: paper analysis, plan generation, plan refinement, file ordering, code writing, and memory. The trained model uses the same scaffold at inference (Figure 2).

Paper analysis and plan generation. Given paper p, independent Concept and Algorithm Agents analyze its method structure and technical details, respectively. A Plan Agent combines their outputs into Q_0, specifying target files, responsibilities, and dependencies in five sections: file structure, implementation components, validation approach, environment setup, and implementation strategy.

Two-pass plan refinement. We correct omissions, simplifications, and misinterpretations in Q_0 through two passes (Figure 4). *Pass 1: analysis-driven global correction.* The refiner checks all five sections against the concept and algorithm analyses. A revision becomes Q_1 only if all sections remain present, the YAML is well formed, and it is no shorter than Q_0; failed or rejected revisions leave $Q_1 = Q_0$. *Pass 2: paper-driven evidence grounding.* For each section and sub-part of Q_1, BM25 retrieves paper passages with expected-section priors, excluding the abstract, introduction, and related work. Structured evidence JSON links plan statements to source passages and guides constrained enrichment of the four remaining sections, leaving file structure unchanged. Revised sections are retained only if no shorter than their inputs; otherwise, their Q_1 content is preserved. Merging them with the unchanged file structure yields Q_2.

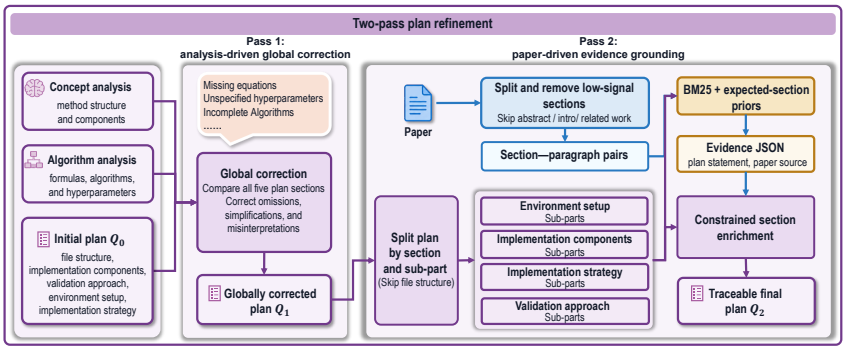

Figure 4. Two-pass plan refinement. Pass 1 applies global analysis-driven correction; Pass 2 audits the plan section by section against retrieved paper evidence and merges only traceable enrichments.

Per-file coding and memory. A structure agent orders the planned files by implementation priority. A code agent writes one file at a time using the plan and accumulated code and memory context; a memory agent summarizes it after each `write_file` action. Subsequent steps access summaries or code through tools. The loop continues until all planned files have been written, yielding the repository and its code memory.

Dataset statistics. The corpus contains 430,428 examples and 16,997.1M tokens under the Sigma tokenizer; Table 2 gives the sub-task breakdown. We mix all six sub-task datasets to fine-tune Sigma MoE (20B total, 0.5B active parameters) (Hu et al., 2025).

3.3 Task-Decomposed RL for Paper-to-Code Reproduction

Collecting full-repository rollouts is costly because the agent must execute the plan pipeline and write interdependent files sequentially before receiving a repository-level reward. This sparse feedback does not distinguish omissions in the plan from implementation errors in individual files. We instead fix sub-task input contexts offline, allowing the plan pipeline and file-scoped code writing to be trained in parallel without full-repository rollouts. Two paper-grounded checklists provide separate rewards for the complete plan and each target file for joint optimization.

Plan checklist construction. For each paper p, we construct $\mathcal{C}_{\text{plan}}(p)$ in two stages. First, an LLM converts implementation-relevant specifications from technical sections and complete configuration tables into atomic, independently verifiable requirements, preserving complete equations, separate procedural steps, and conditions on values. Second, we audit coverage: for entities identified in the paper, we mask their names in requirements and use BM25 to retrieve evidence for missing requirements for other entities; for each implementation section, BM25 retrieves checklist items, which the LLM compares with the section text to identify and recover omitted requirements. Additions require explicit paper support and recorded evidence. Exact and semantic deduplication before and after the audits yields a checklist cached across rollouts (Algorithm 1; example in Appendix F.1).

File-scoped code checklist construction. Given paper p and plan Q, we identify target files from Q's `file_structure`. For each file f, an LLM converts its responsibilities and implementation specifications in Q into atomic, source-code-verifiable requirements, using BM25 to retrieve paper passages only for underspecified details. A self-check against the plan is followed by a paper-anchored critic to identify remaining omissions; only requirements supported by the plan or paper are added. Exact-text deduplication yields $\mathcal{C}_{\text{file}}(p, Q, f)$ (Algorithm 2; example in Appendix F.1.3).

Queries and rewards. A plan query supplies paper p to the plan pipeline. For a target file f, let d_f be its responsibility extracted from Q's `file_structure`, S_f its preceding repository state, and M_f the accumulated code memory. Together with the complete plan Q, these define the file query. Under the shared policy π_θ, the rollouts are

$$\begin{aligned} \widehat{Q} &\leftarrow \text{Rollout}_{\text{plan}}(\pi_\theta, q_{\text{plan}}), \qquad q_{\text{plan}} = p, \\ x_f &\leftarrow \text{Rollout}_{\text{file}}(\pi_\theta, q_{\text{file}}), \qquad q_{\text{file}} = (f, d_f, S_f, M_f, Q). \end{aligned} \tag{2}$$

where each rollout returns either the complete plan $\widehat{Q}$ or code x_f written only to target file f. The file query's input plan Q and preceding context are fixed offline, not generated by the current rollout.

A single generative reward model (GRM) scores each criterion c against artifact x, assigning $s(c, x) = 1$ only for full satisfaction and 0 otherwise, including partial or unverifiable satisfaction. Plans must explicitly specify the required details; code files must implement them. The reward is the fraction of satisfied checklist criteria:

$$r(\mathcal{C}, x) = \frac{1}{|\mathcal{C}|} \sum_{c \in \mathcal{C}} s(c, x),$$
$$r_{\text{plan}} = r(\mathcal{C}_{\text{plan}}(p), \widehat{Q}), \qquad r_{\text{file}} = r(\mathcal{C}_{\text{file}}(p, Q, f), x_f). \tag{3}$$

The checklists are used only for scoring and are not included in the policy inputs. Further details of the GRM scoring protocol are provided in Appendix C.3.

GRPO optimization. Starting from the SFT checkpoint, we jointly optimize the plan and file-writing tasks with GRPO (Shao et al., 2024), using r_{plan} or r_{file} for the corresponding rollout and group-normalizing rewards without task-specific rescaling. Given the highly sparse activation of our MoE backbone, we adopt Rollout Routing Replay (R3) (Ma et al., 2025) to stabilize RL training. Plan examples are sampled five times as often as code examples; the remaining configuration is given in Appendix D. The resulting SFT-RL backbone is the ReSpec model.

4 Experiments

4.1 Experimental Setup

Benchmarks and metrics. We evaluate our method on the 20 papers in PaperBench and all 90 papers in Paper2CodeBench (Starace et al., 2025; Seo et al., 2026). For PaperBench, we report the Code-Dev score computed using its human-authored rubrics. For Paper2CodeBench, we retain the papers but replace its original evaluation protocol with our repository-scoring procedure (Section 3.1). Specifically, instead of assigning five-point ratings to entire repositories, our procedure verifies generated implementation criteria against retrieved code evidence. This yields finer-grained assessments without requiring the complete repository to fit within the judge's context window.

ReSpec and baselines. ReSpec combines the ReSpec model with the reproduction workflow described in Section 3.2. The ReSpec model is a Sigma MoE trained with SFT and RL. We compare ReSpec against three baselines. Basic Agent (Starace et al., 2025) is a general-purpose ReAct-style coding agent that directly constructs a repository from the target paper. PaperCoder (Seo et al., 2026) is a multi-agent framework that organizes paper-to-code reproduction into planning, implementation-specific analysis, and dependency-aware code generation. DeepCode-Base (Li et al., 2025) performs paper analysis, initial plan generation, file ordering, and file-by-file implementation with structured code memory. We evaluate Basic Agent and PaperCoder with o3-mini-high (OpenAI, 2025) and GLM-4.7 (Z.ai, 2025). For DeepCode-Base, we additionally use Claude Sonnet 4.5 (Anthropic, 2025), GLM-4.7-Flash (Z.ai, 2026), Gemma 4-26B-A4B (Gemma Team, 2026), and Qwen3.5 (4B, 9B, and 35B-A3B) (Qwen Team, 2026).

Evaluation and reporting. Following the judge-model configuration of PaperBench and Paper2CodeBench, we use o3-mini-high for benchmark scoring. Benchmark scores are means over three independent runs; reported $\pm$ values denote the standard deviation across runs.

4.2 Main Results

Table 3 compares the complete ReSpec system with the baseline methods on PaperBench Code-Dev and all 90 papers in Paper2CodeBench.

PaperBench Code-Dev. The complete ReSpec system achieves a score of 70.6 with 0.5B active backbone parameters. It exceeds DeepCode-Base with Qwen3.5-35B-A3B by 5.2 points while using one-sixth as many active parameters. It also outperforms the Qwen3.5-9B configuration by 5.1

Method	Backbone	Total	Active	PaperBench	Paper2CodeBench
Basic Agent	o3-mini-high	N/A	N/A	5.1±0.8	-
	GLM-4.7	355B	32B	42.3±0.9	-
PaperCoder	o3-mini-high	N/A	N/A	45.1±0.3	-
	GLM-4.7	355B	32B	52.7±1.5	-
DeepCode-Base	o3-mini-high	N/A	N/A	38.7±0.7	-
	Claude-Sonnet-4.5	N/A	N/A	73.5±2.8	-
	GLM-4.7	355B	32B	**75.0**±2.4	**71.8**±0.1
	GLM-4.7-Flash	30B	3B	60.9±2.1	-
	Gemma4	26B	4B	38.1±2.6	-
	Qwen3.5-4B	4B	4B	53.1±2.7	-
	Qwen3.5-9B	9B	9B	65.5±1.8	55.6±0.4
	Qwen3.5-35B-A3B	35B	3B	65.4±0.7	63.5±0.3
ReSpec	ReSpec model	20B	0.5B	70.6±0.6	68.2±0.4

Table 3. Paper-to-code reproduction scores on PaperBench Code-Dev and criterion-level Paper2CodeBench. Values are means ± standard deviations over three runs; bold marks the best mean in each benchmark. Total and Active denote total and active backbone parameters; N/A denotes undisclosed counts.

points. The two strongest configurations are DeepCode-Base with GLM-4.7 (75.0) and Claude-Sonnet-4.5 (73.5), which outperform ReSpec by 4.4 and 2.9 points, respectively. Thus, ReSpec surpasses all evaluated Qwen3.5-based configurations with substantially fewer active parameters, although it remains behind the strongest baselines. On DeepCode-Base, ReSpec achieves a score of 65.6, corresponding to the "without plan refinement" setting in Table 8; this score still exceeds those of all three evaluated Qwen3.5 backbones.

Paper2CodeBench. Under our criterion-level evaluation protocol, ReSpec achieves a mean repository alignment of 68.2%, compared with 63.5% for DeepCode-Base with Qwen3.5-35B-A3B and 55.6% with Qwen3.5-9B. ReSpec ranks second among the four evaluated configurations, behind DeepCode-Base with GLM-4.7 (71.8%). Results by venue are reported in Appendix E.1.

Overall, synthesized data and process-reward RL enable ReSpec, with 0.5B active parameters, to outperform substantially larger models while approaching Claude-Sonnet-4.5 and GLM-4.7.

4.3 Ablation Study

We assess the effect of RL and plan refinement on the 20 PaperBench Code-Dev papers. Figure 5 shows per-paper score changes; absolute scores and paper IDs are reported in Appendix E.2.

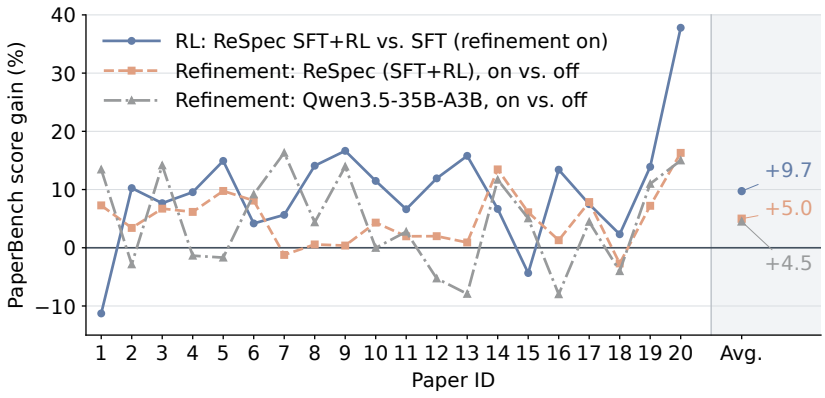

Figure 5. Per-paper changes in PaperBench Code-Dev score. Avg. is the mean over all 20 papers.

Effect of RL. With the inference workflow fixed, including two-pass plan refinement, RL raises the score from 60.9 for the SFT checkpoint to 70.6 for SFT+RL (+9.7 points). Mean scores im-

prove on 18 of 20 papers, with regressions on `adaptive-pruning` and `sapg`. This comparison measures the downstream gain from jointly optimizing the plan and file-writing rewards.

Effect of plan refinement. With model weights fixed, two-pass refinement raises the score from 65.6 to 70.6 for the ReSpec model (+5.0 points) and from 65.4 to 69.9 for the original Qwen3.5-35B-A3B (+4.5 points). Mean scores improve on 18 of 20 papers for ReSpec and 12 of 20 for Qwen3.5-35B-A3B. These gains demonstrate the cross-model effectiveness of two-pass refinement.

4.4 Validation of Repository Scoring

We assess criterion coverage and code-verification accuracy for the repository-scoring procedure used in paper selection and Paper2CodeBench evaluation. The results are summarized in Table 4.

Criterion coverage. We apply the criterion-extraction procedure from Sec. 3.1 to the 20 PaperBench papers, using Qwen3-235B-A22B as the extraction model. Three LLM judges then assess whether the resulting criteria cover each of the 3,671 human-authored Code-Dev rubric items. The three coverage judges obtain recall of 86.4%, 75.8%, and 81.6%, respectively, indicating substantial coverage of the human-authored implementation requirements. Per-paper results appear in Appendix E.

(a) Criterion coverage on PaperBench

Coverage judge	Recall
Qwen3-235B-A22B	86.4
Qwen3.5-397B-A17B	75.8
GLM-4.7	81.6

(b) Code verification on JudgeEval

Procedure / judge	Accuracy
PaperBench	<u>83.0</u>
Ours	
Qwen3-32B	**85.9**
o3-mini-high	<u>85.2</u>
GLM-4.7	84.5
Qwen3.5-9B	70.1

Table 4. Repository-scoring validation and judge selection (%). (a) LLM-assessed coverage of human-authored rubrics. (b) Judge accuracy against human-authored labels.

Verification accuracy. On JudgeEval, our procedure achieves 85.2% accuracy against human-annotated rubric labels, versus 83.0% for the PaperBench procedure, both using o3-mini-high (Table 4(b)). For candidate-file retrieval, PaperBench relies on filenames, whereas our procedure uses the directory tree, code summaries, and dependency graph (Sec. 3.1). These repository views provide structural and semantic cues for locating code relevant to each rubric. The accuracy gain supports the usefulness of this richer context for criterion-level code retrieval and verification.

4.5 Process-Reward Diagnostics

We report checklist coverage and judge-selection evidence for the RL rewards: (1) **File-checklist coverage.** The file-scoped code checklists generated by Qwen3.5-397B-A17B achieve 88.9% rubric-to-checklist recall against Code-Dev rubrics for the 20 PaperBench papers. (2) **Code-GRM selection.** Qwen3-32B achieves the highest accuracy among the evaluated judges (85.9%) on repository-level code verification in JudgeEval (Table 4(b)). We select it for file-scoped code rewards based on this relative ranking. (3) **Plan judgments.** We employ human annotators to assess whether the plans generated for the 20 PaperBench papers satisfy their corresponding generated plan checklists. Using these human criterion-level judgments as the reference, Qwen3-32B achieves a mean accuracy of 87.7% across three evaluations of the same checklist-plan pairs.

5 Conclusion

We presented ReSpec, a framework that combines paper-grounded model training with a structured workflow for paper-to-code reproduction. It selects reproducible papers and automatically constructs intermediate supervision and checklist-based process rewards for SFT and RL.

With 0.5B active backbone parameters, ReSpec achieves a PaperBench score of 70.6 and a mean repository alignment of 68.2 on all 90 Paper2CodeBench papers under criterion-level evaluation.

With the inference workflow fixed, RL improves the SFT checkpoint by 9.7 points on PaperBench. These results support automatically constructed supervision and process rewards as a practical route to improving implementation fidelity in paper-to-code reproduction.

AI USE STATEMENT

In this work, we used generative AI tools for one task with required disclosure: generating synthetic data sets. Our data engine uses large language models (Qwen3.5-397B-A17B as the teacher within the ReSpec scaffold, and Qwen3-235B-A22B and Qwen3-32B for criterion extraction and file ranking) to analyze selected papers, verify candidate repositories, and convert teacher rollouts into supervision for six sub-tasks (Section 3). This use is part of the proposed method itself and is evaluated in the experiments. We did not use generative AI tools for the other tasks with required disclosure: generative AI was not used to develop theoretical models or conceptual frameworks, formulate or prove mathematical claims, propose or refine hypotheses, design or provide feedback on research methodology or experiments, implement the training and evaluation code, translate text, clean or reformat datasets beyond the pipeline described in the paper, support qualitative or thematic data analysis, or interpret results. Additionally, we used generative AI tools (Claude Opus 5 and GPT-6 Astra) for tasks with recommended disclosure, namely editing drafts of the manuscript to improve readability and clarity, and searching, organizing, and summarizing relevant literature. We reviewed all AI-assisted work: the LLM-generated supervision data was filtered through alignment-score and difficulty-aware filtering and checked against paper-grounded checklists (Section 3); every AI-suggested revision to the text was checked by the authors; and references surfaced by AI-assisted literature search were verified against the original publications before citation. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

We use published papers and public GitHub repositories for curation, and teacher models to synthesize supervision from the selected papers. Use and redistribution are subject to applicable licenses and teacher-model terms of use. We release pointers and metadata rather than repackaged third-party code, and restrict released trajectories to a licensable subset.

REPRODUCIBILITY STATEMENT

We release the data engine, plan and file-scoped code checklists with metadata, paper-repository pointers (URLs and commits), a releasable subset of distilled trajectories, and training configurations. Method and evaluation details appear in Sections 3 and 4, with training settings and prompt templates in Appendices D and G, respectively.

REFERENCES

Anthropic. Claude Sonnet 4.5 system card, 2025. URL https://www.anthropic.com/claude-sonnet-4-5-system-card.

Yao Fu, Hao Peng, Litu Ou, Ashish Sabharwal, and Tushar Khot. Specializing smaller language models towards multi-step reasoning. In *International Conference on Machine Learning (ICML)*, volume 202, pp. 10421–10430, 2023. URL https://proceedings.mlr.press/v202/fu23d.html.

Gemma Team. Gemma 4 technical report. *arXiv preprint arXiv:2607.02770*, 2026. URL https://arxiv.org/abs/2607.02770.

Anisha Gunjal, Anthony Wang, Elaine Lau, Vaskar Nath, Yunzhong He, Bing Liu, and Sean Hendryx. Rubrics as rewards: Reinforcement learning beyond verifiable domains. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/cfd7bee7a651ee9af525098ef67a9e45-Abstract-Conference.html.

Qingguo Hu, Zhenghao Lin, Ziyue Yang, Yucheng Ding, Xiao Liu, Yuting Jiang, Ruizhe Wang, Tianyu Chen, Zhongxin Guo, Yifan Xiong, Rui Gao, Lei Qu, Jinsong Su, Peng Cheng, and Yeyun Gong. Sigma-MoE-Tiny technical report. *arXiv preprint arXiv:2512.16248*, 2025. URL https://arxiv.org/abs/2512.16248.

Tianyu Hua, Harper Hua, Violet Xiang, Benjamin Klieger, Sang Truong, Weixin Liang, Fan-Yun Sun, and Nick Haber. ResearchCodeBench: Benchmarking LLMs on implementing novel machine learning research code. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, volume 38, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/cd0d0a873cc3e601c76f46dccc3d4c5f-Abstract-Datasets_and_Benchmarks_Track.html.

Yuxin Jiang, Yufei Wang, Qiyuan Zhang, Xingshan Zeng, Liangyou Li, Jierun Chen, Chaofan Tao, Haoli Bai, and Lifeng Shang. From verifiable dot to reward chain: Harnessing verifiable reference-based rewards for reinforcement learning of open-ended generation. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/1be5bc25d50895ee656b8c2d9eb89d6a-Abstract-Conference.html.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html.

Zongwei Li, Zhonghang Li, Zirui Guo, Xubin Ren, and Chao Huang. DeepCode: Open agentic coding. *arXiv preprint arXiv:2512.07921*, 2025. URL https://arxiv.org/abs/2512.07921.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/aca97732e30bcf1303bc22ac3924fd16-Abstract-Conference.html.

Zijie Lin, Qilin Cai, Liang Shen, and Mingjun Xiao. Enhancing automated paper reproduction via prompt-free collaborative agents. *arXiv preprint arXiv:2512.02812*, 2025a. URL https://arxiv.org/abs/2512.02812.

Zijie Lin, Yiqing Shen, Qilin Cai, He Sun, Jinrui Zhou, and Mingjun Xiao. AutoP2C: An LLM-based agent framework for code repository generation from multimodal content in academic papers. *arXiv preprint arXiv:2504.20115*, 2025b. URL https://arxiv.org/abs/2504.20115.

Tianci Liu, Ran Xu, Tony Yu, Ilgee Hong, Carl Yang, Tuo Zhao, and Haoyu Wang. OpenRubrics: Towards scalable synthetic rubric generation for reward modeling and LLM alignment. *arXiv preprint arXiv:2510.07743*, 2025. URL https://arxiv.org/abs/2510.07743.

Jane Luo, Xin Zhang, Steven Liu, Jie Wu, Jianfeng Liu, Yiming Huang, Yangyu Huang, Chengyu Yin, Ying Xin, Yuefeng Zhan, Hao Sun, Qi Chen, Scarlett Li, and Mao Yang. RPG: A repository planning graph for unified and scalable codebase generation. In *International Conference on Learning Representations (ICLR)*, 2026a. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/9482f45fdd89aba9130bb04c44f788a9-Abstract-Conference.html.

Yujie Luo, Zhuoyun Yu, Xuehai Wang, Yuqi Zhu, Ningyu Zhang, Lanning Wei, Lun Du, Da Zheng, and Huajun Chen. What makes AI research replicable? executable knowledge graphs as scientific knowledge representations. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 841–861, 2026b. doi: 10.18653/v1/2026.acl-short.70. URL https://aclanthology.org/2026.acl-short.70/.

Wenhan Ma, Hailin Zhang, Liang Zhao, Yifan Song, Yudong Wang, Zhifang Sui, and Fuli Luo. Stabilizing MoE reinforcement learning by aligning training and inference routers. *arXiv preprint arXiv:2510.11370*, 2025. URL https://arxiv.org/abs/2510.11370.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pp. 46534–46594, 2023. doi: 10.52202/075280-2019. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

OpenAI. o3-mini, 2025. URL https://platform.openai.com/docs/models/o3-mini.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47717–47737. PMLR, 2025. URL https://proceedings.mlr.press/v267/pan25g.html.

Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Lariviere, Alina Beygelzimer, Florence d'Alche Buc, Emily Fox, and Hugo Larochelle. Improving reproducibility in machine learning research: A report from the NeurIPS 2019 reproducibility program. *Journal of Machine Learning Research*, 22(164):1–20, 2021. URL https://jmlr.org/papers/v22/20-303.html.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. Paper2Code: Automating code generation from scientific papers in machine learning. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/41b6674c28a9b93ec8d22a53ca25bc3b-Abstract-Conference.html.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024. URL https://arxiv.org/abs/2402.03300.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pp. 8634–8652, 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. PaperBench: Evaluating AI's ability to replicate AI research. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267, pp. 56843–56873, 2025. URL https://proceedings.mlr.press/v267/starace25a.html.

Huacan Wang, Ziyi Ni, Shuo Zhang, Shuo Lu, Sen Hu, Ziyang He, Chen Hu, Jiaye Lin, Yifu Guo, Yuntao Du, and Pin Lyu. RepoMaster: Autonomous exploration and understanding of GitHub repositories for complex task solving. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 38, pp. 106320–106359, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/98da9cdb7e8af9192f1fe2cb38690d60-Abstract-Conference.html.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-Shepherd: Verify and reinforce LLMs step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9426–9439, 2024a. doi: 10.18653/v1/2024.acl-long.510. URL https://aclanthology.org/2024.acl-long.510/.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 50208–50232. PMLR, 2024b. URL https://proceedings.mlr.press/v235/wang24h.html.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-Instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 13484–13508, 2023. doi: 10.18653/v1/2023.acl-long.754. URL https://aclanthology.org/2023.acl-long.754/.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with OSS-Instruct. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 52632–52657. PMLR, 2024. URL https://proceedings.mlr.press/v235/wei24h.html.

Yanzheng Xiang, Hanqi Yan, Shuyin Ouyang, Lin Gui, and Yulan He. SciReplicate-Bench: Benchmarking LLMs in agent-driven algorithmic reproduction from research papers. In *Conference on Language Modeling (COLM)*, 2025. URL https://arxiv.org/abs/2504.00255.

Shuo Yan, Ruochen Li, Ziming Luo, Zimu Wang, Daoyang Li, Liqiang Jing, Kaiyu He, Peilin Wu, Juntong Ni, George Michalopoulos, Yue Zhang, Ziyang Zhang, Mian Zhang, Zhiyu Chen, and Xinya Du. LMR-BENCH: Evaluating LLM agent's ability on reproducing language modeling research. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 6164–6186, 2025. doi: 10.18653/v1/2025.emnlp-main.314. URL https://aclanthology.org/2025.emnlp-main.314/.

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 (NeurIPS)*, 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.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 38, 2025. doi: 10.52202/085713-3239. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/8b86cf5ace600c48fd188efbb8dedec8-Abstract-Datasets_and_Benchmarks_Track.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 (ICLR)*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Z.ai. GLM-4.7 model card, 2025. URL https://huggingface.co/zai-org/GLM-4.7.

Z.ai. GLM-4.7-Flash model card, 2026. URL https://huggingface.co/zai-org/GLM-4.7-Flash.

Xuanle Zhao, Zilin Sang, Yuxuan Li, Qi Shi, Weilun Zhao, Shuo Wang, Duzhen Zhang, Xu Han, Zhiyuan Liu, and Maosong Sun. AutoReproduce: Automatic AI experiment reproduction with paper lineage. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 21920–21942, 2026. doi: 10.18653/v1/2026.acl-long.1001. URL https://aclanthology.org/2026.acl-long.1001/.

Mingyang Zhou, Quanming Yao, Lun Du, Lanning Wei, and Da Zheng. Reflective paper-to-code reproduction enabled by fine-grained verification. *arXiv preprint arXiv:2508.16671*, 2025. URL https://arxiv.org/abs/2508.16671.

A Data Curation

A.1 Data Sources

The selected paper pool covers papers published from 2020 through 2025 at the 11 ML and AI venues listed in Table 5.

Abbreviation	Venue
ICLR	International Conference on Learning Representations
ICML	International Conference on Machine Learning
NeurIPS	Conference on Neural Information Processing Systems
CVPR	IEEE/CVF Conference on Computer Vision and Pattern Recognition
ICCV	IEEE/CVF International Conference on Computer Vision
ECCV	European Conference on Computer Vision
ACL	Annual Meeting of the Association for Computational Linguistics
EMNLP	Conference on Empirical Methods in Natural Language Processing
NAACL	Annual Conference of the North American Chapter of the Association for Computational Linguistics
AAAI	AAAI Conference on Artificial Intelligence
ACM MM	ACM International Conference on Multimedia

Table 5. Venues covered by the selected paper pool.

A.2 Curation Thresholds

We group papers by the number of criteria remaining after discarding weight-0 items. The alignment thresholds for the $[0, 150)$, $[150, 300)$, and $[300, +\infty)$ groups are 0.83, 0.80, and 0.77, respectively. Figure 3 shows the candidate score distributions before filtering. The selected pool contains 6,172 paper-repository pairs.

B Reproduction Scaffold and Plan Refinement

B.1 Plan Representation

The scaffold in Section 3.2 uses plans with five substantive sections: `file_structure`, `implementation_components`, `validation_approach`, `environment_setup`, and `implementation_strategy`. An additional `paper_info` field stores paper metadata.

B.2 Tool configuration

Agents use stage-specific tools exposed through the Model Context Protocol (MCP). Paper analysis and initial planning use filesystem tools, whereas first-pass refinement and file ordering use no tools. Second-pass refinement uses `search_paper_bm25` to retrieve paper passages and records paragraph-level evidence for plan revision. The code agent uses `read_file`, `write_file`, and `edit_file` for file operations, `get_file_structure` for repository inspection, and `read_code_mem` to retrieve accumulated code summaries. Each `write_file` action triggers a memory agent to summarize the written file for subsequent coding steps. Web-search, repository-download, and code-execution tools are not exposed to agents in this inference configuration.

B.3 Two-Pass Plan Refinement

The initial plan Q_0 is refined into an intermediate plan Q_1 and then the final plan Q_2 used for code generation.

Pass 1: analysis-driven correction. The refiner compares Q_0 with the concept and algorithm analyses without accessing the paper. Each refinement invocation allows up to three attempts; if none is accepted, $Q_1 = Q_0$.

Pass 2: paper-evidence grounding. The paper is segmented once into indexed, section-labeled passages shared across the four refinable plan sections.

- **Query construction.** The agent is instructed to construct named sub-parts from verbatim plan excerpts and select expected sections from the paper's headings.
- **Retrieval settings.** BM25 uses $k_1 = 1.5$ and $b = 0.75$, with a multiplicative factor of 1.5 for expected-section matches. For each sub-part, up to eight positive-scoring, non-heading passages are selected. These are deduplicated across sub-parts and pruned to a shared 10,000-token budget per retrieval call, removing passages with the lowest maximum score across matched sub-parts. Retained passages are returned in document order.
- **Section dependencies.** The `implementation_components`, `validation_approach`, and `environment_setup` sections are refined in parallel, with `file_structure` supplied as shared context. The `implementation_strategy` section is refined afterward, additionally using the currently retained `implementation_components` section.
- **Evidence recording.** Before returning the revised section, the agent is instructed to record supporting paragraph indices, section names, retrieval scores, and excerpts in a JSON mapping.

Each revised section is retained only if it is no shorter than its input; otherwise, its Q_1 content is preserved. Merging these sections with the unchanged `file_structure` yields Q_2.

The refinement prompts are provided in Appendix G.1.3, and Appendix F.2 illustrates both passes.

C Checklist-Based Process Rewards

Checklist construction is separate from candidate scoring. A plan checklist uses only the paper and its derived artifacts; a file-scoped checklist also uses the generated plan and its assignment of file responsibilities. Neither constructor reads candidate code. The GRM subsequently evaluates the candidate plan or file against the corresponding checklist.

C.1 Plan Checklist Construction

Algorithm 1 constructs a paper-only checklist cached across plan rollouts. Extraction preserves complete equations, individual procedure steps, and conditions on values. Entity expansion retrieves paper passages, whereas section auditing retrieves existing checklist items. Both audits add only paper-supported requirements and record their evidence. The prompts are provided in Appendix G.2.

Algorithm 1: Plan checklist construction

Input: paper p
Output: cached checklist $\mathcal{C}_{\text{plan}}(p)$ and evidence log L

Split p by headings into sections S and identify its entities X.
Extract and standardize atomic requirements C_0 from technical sections and complete configuration tables, using X.
$C_0 \leftarrow \text{SemanticDedup}(\text{ExactDedup}(C_0))$
Group requirements by templates with entity names masked; identify missing entity coverage using X.
Retrieve paper passages with BM25 to verify missing entity-specific requirements; record additions A_{ent} and evidence L_{ent}.
$C_1 \leftarrow \text{ExactDedup}(C_0 \cup A_{\text{ent}})$
for each implementation-relevant section $s \in S$ **do**
    Retrieve relevant items B_s from C_1 using BM25.
    Compare s with B_s to produce missing requirements A_s and their evidence L_s.
end for
$C_2 \leftarrow \text{ExactDedup}(C_1 \cup \bigcup_s A_s)$; $L \leftarrow L_{\text{ent}} \cup \bigcup_s L_s$
Cache and **return** $\mathcal{C}_{\text{plan}}(p) = \text{SemanticDedup}(C_2)$ and L.

C.2 File-Scoped Code Checklist Construction

Algorithm 2 converts each file's responsibilities in the complete plan Q into code-verifiable requirements. Paper retrieval resolves underspecified details; a self-check and a paper-anchored critic recover omissions. Only requirements supported by Q or p are retained, without consulting candidate code. The prompts are provided in Appendix G.3.

Algorithm 2: File-scoped code-checklist construction

Input: paper p and generated reproduction plan Q
Output: a checklist $\mathcal{C}_{\text{file}}(p, Q, f)$ for each target file f

$\mathcal{F} \leftarrow \text{ExtractTargetFiles}(Q.\texttt{file_structure})$
$R_p \leftarrow \text{BuildBM25Index}(\text{Passages}(p))$
for each $f \in \mathcal{F}$ **do**
  Draft atomic requirements D_f from f's responsibilities in Q; query R_p only for underspecified details.
  Self-check D_f against Q to recover omissions, yielding $C_f^{(1)}$.
  Use a paper-anchored critic with Q and R_p to identify requirements ΔC_f missing from $C_f^{(1)}$.
  $\mathcal{C}_{\text{file}}(p, Q, f) \leftarrow \text{ExactTextDedup}(C_f^{(1)} \cup \Delta C_f)$
end for
return $\{f : \mathcal{C}_{\text{file}}(p, Q, f) \mid f \in \mathcal{F}\}$

C.3 GRM Scoring Protocol

For each judgment, the same GRM receives one atomic criterion c together with the complete candidate plan $\widehat{Q}$ or the complete code x_f written to target file f, and returns a JSON verdict with a short justification. Judgments rely only on the supplied plan or file. Plans must explicitly state the required values, formulas, and scope.

For code files, we use `FILE_CRITERION_JUDGMENT_PROMPT`. The GRM checks implementation logic in the relevant pipeline stage, not identifier names or comments alone. Required values may be set directly or exposed as configurable options; conflicting hard-coded values with no way to change them are rejected. Standard engineering or theoretical equivalences are accepted. Missing, incorrect, incomplete, or unverifiable implementations, including placeholders and broken execution paths, receive zero.

The `verdict` field uses the strings `satisfied` or `not_satisfied` for plans, and Boolean `true` or `false` for code files. Both formats map to the same binary score: $s(c, x) = 1$ only for full satisfaction and $s(c, x) = 0$ otherwise, including partial or unverifiable cases.

The rewards r_{plan} and r_{file} are computed separately as the unweighted mean of the criterion scores in their respective checklists. Checklists are used only for scoring and are not included in policy inputs. The scoring prompts are provided in Appendix G.4.

Algorithm 3: Task-decomposed RL with checklist rewards

Input: SFT-initialized policy π_θ; offline records $\mathcal{D}$; rollout group size G
Output: jointly optimized policy π_θ

for each training step **do**
  Sample a batch $\mathcal{B}$ from $\mathcal{D}$, drawing plan examples five times as often as code examples; initialize an empty buffer $\mathcal{T}$.
  for each $(\tau, q_\tau, \mathcal{C}_\tau) \in \mathcal{B}$ **do**
    Generate G task-τ rollouts under π_θ with input q_τ; retain trajectories ζ_j and artifacts x_j.
    $r_j \leftarrow \frac{1}{|\mathcal{C}_\tau|} \sum_{c \in \mathcal{C}_\tau} s(c, x_j)$ for $j = 1, \ldots, G$
    Compute advantages $\widehat{A}_j$ by group-normalizing $\{r_j\}_{j=1}^{G}$ within this query, without task-specific rescaling.
    Append $(q_\tau, \zeta_j, \widehat{A}_j)$ for $j = 1, \ldots, G$ to $\mathcal{T}$.
  end for
  Update π_θ with GRPO on $\mathcal{T}$, using Rollout Routing Replay (R3).
end for
return π_θ

C.4 TASK-DECOMPOSED RL

Algorithm 3 uses offline records $(\tau, q_\tau, \mathcal{C}_\tau)$, pairing each task and query from Eq. 2 with its checklist. File queries use a fixed offline plan and preceding context, not outputs of current plan rollouts. We retain each rollout's trajectory ζ_j for training and score its artifact x_j: either the complete plan $\widehat{Q}$ or code x_f written only to target file f. Checklists are available only to the GRM, not to the policy.

D TRAINING CONFIGURATION

D.1 TRAINING DATA AND INITIALIZATION

The SFT backbone is trained on the six sub-task datasets described in Section 3.2. RL initializes the actor from this SFT checkpoint. The RL corpus contains 76,448 samples across 2,582 unique projects, with approximately 17.6 checklist criteria per case.

D.2 OPTIMIZATION SETTINGS

Starting from the SFT checkpoint, GRPO jointly optimizes the shared policy for the plan pipeline and file-scoped code writing. Both task rewards lie in $[0, 1]$ and are normalized within each query's rollout group without task-specific rescaling. Plan examples are sampled five times as often as code examples. Rollout Routing Replay (R3) stabilizes training of the sparsely activated MoE backbone. Table 6 summarizes the reported backbone and training settings. Additional data-scale, filtering, and hardware details will accompany the released dataset card.

Parameter	Value
Total / Activated Parameters	20B / 0.5B active
RL Training Steps	300
N-Samples per Query	32
Clipping Range	[0.8, 1.2]
Query Batch Size	64
Mini Batch Size	2,048
Total Batch Size	2,048
Rollout Temperature	1.0
Max Number of Turns	30
Max Tokens per Turn	32768
Max Total Length	131072
Plan:Code Sampling Ratio	5:1
Optimizer and Learning Rate	AdamW & 1e-6

Table 6. Reported backbone and training settings.

E SUPPLEMENTARY EVALUATION

E.1 PAPER2CODEBENCH RESULTS BY VENUE

Table 7 reports criterion-level alignment scores by venue for all 90 Paper2CodeBench papers. The papers contain 33,005 extracted reproduction criteria, averaging 366.7 per paper. The ICLR, ICML, and NeurIPS 2024 subsets each contain 30 papers and contribute 11,179, 11,986, and 9,840 criteria, respectively.

Method / backbone	ICLR 2024	ICML 2024	NeurIPS 2024	Overall
GLM-4.7	70.7±0.4	71.0±0.3	73.8±0.1	71.8±0.1
Qwen3.5-35B-A3B	62.0±0.5	64.5±0.6	64.1±0.2	63.5±0.3
Qwen3.5-9B	56.4±0.5	52.9±0.3	57.5±0.8	55.6±0.4
ReSpec	69.3±0.6	66.5±0.4	68.8±0.4	68.2±0.4

Table 7. Paper2CodeBench criterion-level alignment scores by venue (%). Results are reported as mean ± sample standard deviation across three runs. Each venue contributes 30 papers. All comparison backbones use the DeepCode-Base scaffold; ReSpec denotes the complete system with two-pass plan refinement.

E.2 Component Analysis

Table 8 gives the paired paper-level results used in Figure 5. ReSpec variants differ in RL training and two-pass plan refinement. Both Qwen3.5-35B-A3B variants use the original backbone weights and differ only in refinement.

Paper	ReSpec			Qwen3.5-35B-A3B	
	SFT Ref. on	SFT+RL Ref. off	SFT+RL Ref. on	Original Ref. off	Original Ref. on
adaptive-pruning	58.8±11.3	40.3±3.3	47.5±3.3	42.2±7.6	55.7±10.1
all-in-one	53.9±8.1	60.8±4.8	64.2±3.5	71.2±5.8	68.4±12.4
bam	76.5±2.8	77.5±11.3	84.2±4.5	70.6±12.5	84.7±10.1
bbox	66.3±11.2	69.7±6.7	75.9±3.5	60.6±5.8	59.3±8.8
bridging-data-gaps	47.1±4.1	52.2±10.9	62.0±3.2	58.6±11.0	56.9±4.4
fre	75.3±5.0	71.4±2.4	79.5±3.9	78.4±3.2	87.6±2.6
ftrl	48.2±6.9	55.1±6.1	53.8±1.7	51.3±5.8	67.6±1.5
lbcs	78.0±11.1	91.5±1.8	92.1±0.5	80.5±1.0	84.9±2.0
lca-on-the-line	52.9±15.3	69.1±3.0	69.5±5.9	50.3±15.9	64.3±0.0
mechanistic-understanding	72.2±9.2	79.4±5.2	83.7±4.0	81.3±7.7	81.3±15.8
pinn	76.9±9.8	81.5±5.1	83.5±4.1	73.2±4.0	75.9±2.8
rice	65.9±11.6	75.9±7.3	77.9±1.1	77.1±5.0	71.8±3.1
robust-clip	44.1±4.2	59.0±1.2	59.9±11.5	54.7±6.5	46.8±21.6
sample-specific-masks	55.3±5.4	48.5±5.6	62.0±9.7	45.7±6.4	57.4±17.7
sapg	47.4±17.7	36.9±0.9	43.0±0.6	61.1±19.3	66.1±14.4
sequential-neural-score-est.	62.9±13.7	75.0±1.9	76.3±8.5	81.0±2.3	73.1±20.0
stay-on-topic-with-cfg	60.6±7.0	60.3±4.0	68.1±5.8	66.7±3.7	71.2±0.0
stochastic-interpolants	72.2±5.6	77.2±2.7	74.5±3.6	81.1±2.5	77.1±1.6
test-time-model-adaptation	64.1±4.7	70.8±7.5	78.0±1.5	61.5±13.6	72.5±3.6
what-will-my-model-forget	39.2±1.9	60.7±8.5	77.0±2.4	60.6±8.0	75.6±1.9
Average	60.9±8.3	65.6±0.4	70.6±0.6	65.4±0.7	69.9±0.4

Table 8. PaperBench Code-Dev component analysis. Entries report average scores and standard deviations over three runs. For the Average row, scores are first averaged across the 20 papers within each run. Ref. denotes two-pass plan refinement. Original denotes the Qwen3.5-35B-A3B checkpoint without our SFT or RL. The Sigma SFT+RL checkpoint is the ReSpec model; with refinement enabled, it forms the complete ReSpec system. Paper order matches Figure 5.

Pretrained backbone. The pretrained Sigma backbone, without our SFT or RL, scores 0.0 on PaperBench Code-Dev in the DeepCode-Base scaffold.

E.3 Repository-Scoring Validation

Table 9 reports coverage by paper for the criteria generated with Qwen3-235B-A22B. Each coverage judge evaluates the same generated criteria against the human Code-Dev rubric. Macro recall weights papers equally; micro recall weights each paper by its rubric count.

Paper	Rubrics	Qwen3 235B-A22B	Qwen3.5 397B-A17B	GLM-4.7
adaptive-pruning	86	91.9	83.7	90.7
all-in-one	92	88.0	80.4	87.0
bam	255	92.2	82.4	83.1
bbox	145	66.2	60.0	60.0
bridging-data-gaps	52	82.7	71.2	71.2
fre	306	85.0	76.8	75.5
ftrl	120	81.7	80.8	81.7
lbcs	485	91.3	74.4	87.4
lca-on-the-line	403	83.9	57.8	78.2
mechanistic-understanding	36	97.2	80.6	91.7
pinn	126	96.8	85.7	92.9
rice	178	73.6	63.5	69.1
robust-clip	70	84.3	65.7	74.3
sample-specific-masks	87	83.9	52.9	75.9
sapg	77	90.9	83.1	90.9
sequential-neural-score-estimation	67	91.0	91.0	89.6
stay-on-topic-with-cfg	70	84.3	71.4	75.7
stochastic-interpolants	58	93.1	79.3	82.8
test-time-model-adaptation	86	86.0	82.6	83.7
what-will-my-model-forget	872	84.3	92.2	90.4
Micro recall	3,671	85.7	77.5	82.9
Macro recall		86.4	75.8	81.6

Table 9. Per-paper rubric coverage (%). Qwen3-235B-A22B is the fixed criterion generator; the three rightmost columns identify coverage judges, not generators.

F Case Studies

The following cases illustrate reproduction criteria and checklists for one paper, and two-pass plan refinement for another. They show selected artifacts and their relationship to source evidence.

F.1 Reproduction Criteria and Checklists

For *Graph Filtration Learning* (ICML 2020), we show the weighted reproduction criteria used for paper selection (Section 3.1), the paper-only plan checklist, and a file-scoped code checklist (Appendix C). The excerpts mark omissions with "`...`". Long lines are wrapped, and mathematical symbols in the JSON and Markdown sources are transliterated to ASCII.

F.1.1 Reproduction Criteria

Curation extracts 92 atomic criteria and assigns each an importance weight $w(c) \in \{1, \ldots, 4\}$. The counts at weights 1, 2, 3, and 4 are 20, 5, 26, and 41, respectively. Verification against the paired reference repository marks 83 criteria as implemented, yielding $A_{\text{repo}} = 0.904$ under Eq. 1. Each record contains a criterion, its weight, and the reason for that weight.

Extracted criteria (`criteria_with_scores.json`), records 1 and 6 of 92.

```
[
  {
    "criterion": "Graphs are treated as 1-dimensional simplicial
                  complexes in the context of persistent homology.",
    "importance": 3,
    "reason": "This statement defines a foundational structural
               assumption in the implementation of persistent
               homology on graphs, which is core logic (Score 3). ..."
  },
  ...
  {
    "criterion": "The multiplicity of topological features is
                  encoded using the formula mu_k^{i,j} =
                  (beta_k^{i,j-1} - beta_k^{i,j}) -
                  (beta_k^{i-1,j-1} - beta_k^{i-1,j}) for
                  1 <= i < j <= m.",
    "importance": 4,
    "reason": "This formula represents a core contribution
               implementation (Score 4) as it defines the
               computation of the multiplicity of topological
               features using persistent homology ..."
  },
  ...
]
```

Verification adds a binary verdict x_i and supporting code evidence. These verdicts assess the paired reference repository, not generated candidate code. Of the three artifacts shown here, only the weighted reproduction criteria are used for paper selection.

Verification record, same criterion as record 1 above.

```
{
  "criterion": "Graphs are treated as 1-dimensional simplicial
                complexes in the context of persistent homology.",
  "matched_files": ["core/model.py", "core/data.py",
                    "core/train_engine.py"],
  "importance": 3,
  "verdict": true,
  "evidence": "The code explicitly implements graphs as
     1-dimensional simplicial complexes by computing persistent
     homology on vertices (0-simplices) and edges (1-simplices).
     In `core/data.py`, the function `get_boundary_info` extracts
     the edge list of the graph, which represents the 1-simplices.
     In `core/model.py`, the `PershomBase.forward` method prepares
     the input for the persistence backend by pairing vertex
     filtration values (`node_filt`) with the boundary information
     (`batch.boundary_info`) ...
}
```

F.1.2 PLAN CHECKLIST

The cached paper-only checklist $\mathcal{C}_{\text{plan}}(p)$ for the same paper contains 282 requirements, each written as a separate line with a section tag and checkable against a candidate plan alone. The checklist is built once without reading a candidate plan and reused across plan rollouts. The excerpts retain the recorded tags, including `unclassified/unclassified`.

The following paper passage provides the context for three checklist items: the choice of a weighting function, its formula, and its learnable parameters.

Paper, Section 4.1, "Selection of ν".

```
Selection of nu. We use a vectorization based on a local weighting
function, s : R^2 -> R, of points in a given barcode, B. In
particular, we select the rational hat structure element of (Hofer
et al., 2019b), which is of the form

    B (ni) p |-> 1/(1 + ||p - c||_1) - 1/(1 + | |r| - ||p - c||_1 |) ,

where c in R^2, r in R are learnable parameters. Intuitively, this
can be seen as a function that evaluates the "centrality" of each
point p of the barcode B w.r.t. a learnt center c and a learnt
shift/radius r.
```

The checklist separates these requirements into individually scored items. The excerpt also includes requirements for the barcode representation and its use in a downstream layer.

Plan checklist `C_plan(p)`, lines 1-3 and 8-10 of 282.

```
- [unclassified/unclassified] The barcode coordinate function V is
  defined as V : B -> R mapping a barcode B to
  sum_{(b,d) in B} s(b,d), where s : R^2 -> R is a differentiable
  function that vanishes on the diagonal of R^2.
- [unclassified/unclassified] The model uses d barcode coordinate
  functions to map a barcode into R^d.
- [unclassified/unclassified] The resulting R^d barcode
  representation is fed through a differentiable downstream layer,
  such as a classifier.
...
- [unclassified/unclassified] The rational hat structure element
  from (Hofer et al., 2019b) is selected as the local weighting
  function for the vectorization nu.
- [unclassified/unclassified] The rational hat structure element is
  defined by the formula B (ni) p |-> 1/(1 + ||p - c||_1) -
  1/(1 + | |r| - ||p - c||_1 |).
- [unclassified/unclassified] In the rational hat structure element
  formula, c in R^2 and r in R are learnable parameters.
```

Configuration details are likewise expressed as separate requirements with their applicable scope. The following excerpt pairs the paper's prose with five checklist items.

Paper, Section 5.1 (abridged), and five corresponding checklist items.

```
First, to implement the filter function f, we use a single GIN-eps
layer of (Xu et al., 2019) for one level of message passing (i.e.,
1-GIN) with hidden dimensionality of 64. The obtained latent node
representation is then passed through a two layer MLP, mapping from
R^64 -> [0,1], with a sigmoid activation at the last layer. Node
degrees and (if available) discrete node attributes are encoded via
embedding layers with 64 dimensions. ...

- [unclassified/unclassified] A single GIN-eps layer (1-GIN) from
  (Xu et al., 2019) is used to implement the filter function f for
  one level of message passing.
- [unclassified/unclassified] The GIN-eps layer (1-GIN) uses a
  hidden dimensionality of 64.
- [unclassified/unclassified] The latent node representation
  obtained from the GIN-eps layer is passed through a two-layer MLP.
- [unclassified/unclassified] Node degrees are encoded via embedding
  layers with 64 dimensions.
- [unclassified/unclassified] Discrete node attributes, if
  available, are encoded via embedding layers with 64 dimensions.
```

F.1.3 File-Scoped Code Checklist

Each file-scoped checklist $\mathcal{C}_{\text{file}}(p, Q, f_i)$ is derived from the responsibilities that the final plan Q assigns to target file f_i. For this paper, we show $f_i =$ `models/persistent_homology.py`. The plan names this file in `file_structure` and assigns it two components: filtration construction and persistent-homology computation.

Generated plan Q, `file_structure` entry and components 2 and 3.

```
file_structure:
  graph_filtration_learning/
  ...
  |- models/
  |   |- persistent_homology.py    # PH computation (Component 3):
  |                                 # mu_k^{i,j} formula, O(m^3) complexity
  ...

COMPONENT 2: Sub-Levelset Filtration Constructor
             [models/persistent_homology.py]
- Purpose: Build filtration sequence K^{f,i} based on filter values
  for persistent homology computation
- Input: Graph G as simplicial complex K_G, filter values
```

```
  {f(v) : v in V}
- Process:
  1. Compute filter values for all vertices: {f(v) : v in V}
  2. Sort vertices by filter values: a_1 < a_2 < ... < a_m
     where m = |V|
  3. For each threshold a_i, add all simplices sigma where
     max_{v in sigma} f(v) <= a_i
- Formula: K^{f,i} = {sigma in K : max_{v in sigma} f(v) <= a_i}
  (Eq. 5)
  - K^{f,0} = {}
- Output: Sequence of simplicial complexes forming filtration
- Superlevel alternative: Use g(v) = -f(v) for super-levelset
  filtration (complementary information)

COMPONENT 3: Persistent Homology Computer
             [models/persistent_homology.py]
- Purpose: Compute persistence barcodes B_k from filtration,
  capturing topological features
- Algorithm: Parallel GPU variant of Edelsbrunner et al. (2002)
  matrix reduction (adapted from Hofer et al., 2019a)
...
- Formula: mu_k^{i,j} = (beta_k^{i,j-1} - beta_k^{i,j}) -
  (beta_k^{i-1,j-1} - beta_k^{i-1,j}) (Eq. 4)
- Output: Persistence barcodes with multiplicities
  B_k = {(birth, death, multiplicity)}
...
```

The checklist names these components and translates their specifications into atomic requirements. For Component 2, the *Input* yields two items, *Process* step 2 specifies the filter-value sort order, and the *Formula* yields the item citing Eq. 5.

File-scoped checklist `C_file(p, Q, models/persistent_homology.py)`, lines 1-14 and 18-19 of 26.

```
### File: `graph_filtration_learning/models/persistent_homology.py`
**Purpose:** Compute persistent homology barcodes from graph
filtrations including sub-levelset and super-levelset constructions
with GPU-accelerated matrix reduction.

**Checklist:**
- File implements sub-levelset filtration constructor
  functionality (Component 2)
- File implements persistent homology computer functionality
  (Component 3)
- Function accepts graph G as simplicial complex K_G as input
- Function accepts filter values {f(v) : v in V} for all vertices
  as input
- Function outputs sequence of simplicial complexes forming the
  filtration
- Function outputs persistence barcodes B_k as
  (birth, death, multiplicity) tuples
- Filtration construction sorts vertices by filter values:
  a_1 < a_2 < ... < a_m where m = |V|
- Filtration adds simplices sigma where
  max_{v in sigma} f(v) <= a_i for each threshold a_i
- Filtration formula K^{f,i} = {sigma in K :
  max_{v in sigma} f(v) <= a_i} is implemented (Eq. 5)
- Initial filtration state K^{f,0} = {} is implemented
- Super-levelset filtration uses g(v) = -f(v) for reverse
  filtration
...
- Persistence pairing formula mu_k^{i,j} =
  (beta_k^{i,j-1} - beta_k^{i,j}) -
  (beta_k^{i-1,j-1} - beta_k^{i-1,j}) is implemented (Eq. 4)
- Boundary matrix construction d_k : K_k -> C_{k-1}(K) is
  implemented
...
- Code computes both essential and non-essential features for
  0-dim and 1-dim
```

The checklist retains the plan's filtration equation (Eq. 5), pairing formula (Eq. 4), and sub-levelset and super-levelset requirements. When the plan omits a code-verifiable detail, the constructor consults the paper. If neither source specifies the detail, no item is added.

F.2 TWO-PASS PLAN REFINEMENT

We follow a plan for *Sample-specific Masks for Visual Reprogramming-based Prompting* through both refinement passes. Its YAML representation uses the five substantive sections and `paper_info` metadata described in Appendix B. We show selected excerpts and the evidence supporting each revision.

Pass 1: analysis-driven correction. Pass 1 reads Q_0 and the concept and algorithm analyses, without accessing the paper. It replaces placeholders and vague statements with details from the analyses and labels unresolved choices as defaults. The `file_structure` excerpts below illustrate added configuration and architecture details.

`file_structure` entries, Q_0 and Q_1 (abridged).

```
Q_0:  default_config.yaml  # Default hyperparameters
      mask_generator.py    # 5-layer & 6-layer CNN (PRIORITY 1)
        Layer 2: Conv2d(64->64, k=3, s=1, p=1) + ReLU
                 -> MaxPool2d(k=2, s=2)

Q_1:  default_config.yaml  # epochs=200, batch_size=256, lr=0.01,
                           # milestones=[100,145], decay=0.1
      mask_generator.py    # 5-layer & 6-layer CNN (PRIORITY 1):
                           # 26,499 params (ResNet), 102,339 (ViT),
                           # output floor(H/8) x floor(W/8) x 3
        Layer 2: Conv2d(64->64, k=3, s=1, p=1) + ReLU
                 -> MaxPool2d(k=2, s=2, p=0)
```

Pass 1 also expands the frequent-label-mapping summary into the two algorithms extracted by the analysis: construction of a frequency matrix $d \in \mathbb{Z}^{|Y^{\mathrm{P}}| \times |Y^{\mathrm{T}}|}$ and greedy assignment. It adds the minimization objective of Eq. 4 and its symbol definitions. Choices left unspecified by the analyses are marked as defaults, such as "Xavier/Kaiming uniform (suggested default, not specified in paper)".

Pass 2: paper-evidence grounding. Pass 2 treats the four sections other than `file_structure` independently. Each section is split into named sub-parts, which query the paper's 133 labeled paragraphs using BM25 with expected-section priors. The retrieved excerpts then guide section edits. In `validation_approach`, the analyses supplied no ablation values, so Pass 1 used estimated ranges. Retrieval of Table 3 in Section 5 replaces these ranges with the paper's reported averages. These are validation references from the paper, not results obtained by executing the generated repository.

Ablation entries in `validation_approach`, Q_1 and Q_2 (abridged).

```
Q_1:  ONLY delta     | ~48-50% | Good for feature-rich datasets
      ONLY f_mask    | ~45-47% | Suboptimal with enough training data
      Single-channel | ~42-44% | Less effective, limited color palettes
      SMM (full)     | ~52-54% | Best performance

Q_2:  ONLY delta     | 46.85%  | Good for feature-rich datasets (Table 3)
      ONLY f_mask    | 42.59%  | Suboptimal with enough training data
      Single-channel | 49.70%  | Less effective, limited color palettes
      SMM (full)     | 52.53%  | Best performance
```

Retrieval also corrects figure references. A training-dynamics entry cites "Figure 2: Training Loss Distribution", but the retrieved passage from Section D.2 identifies the training accuracy and loss curves as Figure 11. Pass 2 updates the reference accordingly. Table 10 summarizes representative edits from both passes.

Pass	Grounding signal	Representative edit
1	Algorithm analysis	Flm summary → Algorithms 2-3 (frequency matrix, greedy assignment)
1	Concept analysis	Undetermined choices flagged as defaults (Xavier/Kaiming init)
2	Table 3, Section 5	Ablation ranges → reported averages (46.85 / 42.59 / 49.70 / 52.53)
2	Appendix A.3	ViT-B32 rows added to the interpolation-efficiency comparison
2	Section D.2	"Figure 2" → Figure 11 for the training-curve check

Table 10. Representative plan edits and the signal each is grounded in.

G PROMPTS USED BY THE DATA ENGINE AND THE REWARD PIPELINE

This appendix presents the prompts grouped by pipeline stage. The SFT sub-task prompts (Appendix G.1.1– G.1.5) follow the DeepCode decomposition the data engine adopts, with the addition of the two plan-refinement passes; the checklist-construction prompts (Appendix G.2.1–G.3) and the GRM prompts (Appendix G.4) are the ones introduced by ReSpec for reward construction. Decorative markers (emoji, check marks, and rule lines) and Markdown emphasis markers have been removed. GRM templates use the binary verdict schema defined in Section 3.3; where a template contains `{manifest}` or `{specification}`, that field is filled at call time. Inputs depend on the stage. Plan-checklist construction uses only the paper and its derived artifacts. File-checklist construction additionally receives the target plan, but neither checklist pipeline reads candidate code or the paired reference repository. SFT prompts consume the paper and generated analyses, plans, code, or memory as appropriate. The GRM reads the candidate being scored, while curation verification reads the paired reference repository.

G.1 SUPERVISED SUB-TASK PROMPTS

G.1.1 PAPER ANALYSIS

Two agents analyze the paper independently. The concept analysis prompt maps the paper structure and the components that must be implemented; the algorithm analysis prompt extracts the fine-grained technical content (pseudocode, formulas, hyperparameters).

Concept analysis prompt.

```
PAPER_CONCEPT_ANALYSIS_PROMPT

You are doing a COMPREHENSIVE analysis of a research paper to understand its complete
     structure, contributions, and implementation requirements.

# OBJECTIVE
Map out the ENTIRE paper structure and identify ALL components that need implementation for
     successful reproduction.

# DOCUMENT READING STRATEGY

## TRADITIONAL APPROACH: Complete Document Analysis
Read the entire document systematically to ensure comprehensive understanding:

## EFFICIENT READING -- read each part once, never re-read
Always read by explicit line range. To read a long paper WITHOUT exhausting the
context window:
1. Call `get_file_info(path)` first to learn the file's `total_lines`.
2. Read the paper in SEQUENTIAL, NON-OVERLAPPING ranges with
   `read_text_file(path, start_line=<a>, end_line=<b>)` -- e.g. 1-200, then
   201-400, then 401-600, ... until you reach `total_lines`.
3. Each result is prefixed with `[lines X-Y of N]`; use it to choose the NEXT
   range to read.
4. Do NOT re-read a range you have already read -- re-reading the same lines
   only wastes the context window and gives you nothing new.

# COMPREHENSIVE ANALYSIS PROTOCOL

## 1. COMPLETE PAPER STRUCTURAL ANALYSIS
Create a full map of the document:
```

```yaml
paper_structure_map:
  title: "[Full paper title]"

  sections:
    1_introduction:
      main_claims: "[What the paper claims to achieve]"
      problem_definition: "[Exact problem being solved]"

    2_related_work:
      key_comparisons: "[Methods this work builds upon or competes with]"

    3_method:  # May have multiple subsections
      subsections:
        3.1: "[Title and main content]"
        3.2: "[Title and main content]"
      algorithms_presented: "[List all algorithms by name]"

    4_experiments:
      environments: "[All test environments/datasets]"
      baselines: "[All comparison methods]"
      metrics: "[All evaluation metrics used]"

    5_results:
      main_findings: "[Key results that prove the method works]"
      tables_figures: "[Important result tables/figures to reproduce]"
```

2. METHOD DECOMPOSITION
For the main method/approach:

```yaml
method_decomposition:
  method_name: "[Full name and acronym]"

  core_components:  # Break down into implementable pieces
    component_1:
      name: "[e.g., State Importance Estimator]"
      purpose: "[Why this component exists]"
      paper_section: "[Where it's described]"

    component_2:
      name: "[e.g., Policy Refinement Module]"
      purpose: "[Its role in the system]"
      paper_section: "[Where it's described]"

  component_interactions:
    - "[How component 1 feeds into component 2]"
    - "[Data flow between components]"

  theoretical_foundation:
    key_insight: "[The main theoretical insight]"
    why_it_works: "[Intuitive explanation]"
```

3. IMPLEMENTATION REQUIREMENTS MAPPING
Map paper content to code requirements:

```yaml
implementation_map:
  algorithms_to_implement:
    - algorithm: "[Name from paper]"
      section: "[Where defined]"
      complexity: "[Simple/Medium/Complex]"
      dependencies: "[What it needs to work]"

  models_to_build:
    - model: "[Neural network or other model]"
      architecture_location: "[Section describing it]"
      purpose: "[What this model does]"

  data_processing:
    - pipeline: "[Data preprocessing needed]"
      requirements: "[What the data should look like]"

  evaluation_suite:
    - metric: "[Metric name]"
      formula_location: "[Where it's defined]"
      purpose: "[What it measures]"
```

```
## 4. EXPERIMENT REPRODUCTION PLAN
Identify ALL experiments needed:

```yaml
experiments_analysis:
  main_results:
    - experiment: "[Name/description]"
      proves: "[What claim this validates]"
      requires: "[Components needed to run this]"
      expected_outcome: "[Specific numbers/trends]"

  ablation_studies:
    - study: "[What is being ablated]"
      purpose: "[What this demonstrates]"

  baseline_comparisons:
    - baseline: "[Method name]"
      implementation_required: "[Yes/No/Partial]"
      source: "[Where to find implementation]"
```

5. CRITICAL SUCCESS FACTORS
What defines successful reproduction:

```yaml
success_criteria:
  must_achieve:
    - "[Primary result that must be reproduced]"
    - "[Core behavior that must be demonstrated]"

  should_achieve:
    - "[Secondary results that validate the method]"

  validation_evidence:
    - "[Specific figure/table to reproduce]"
    - "[Qualitative behavior to demonstrate]"
```

OUTPUT FORMAT
```yaml
comprehensive_paper_analysis:
  executive_summary:
    paper_title: "[Full title]"
    core_contribution: "[One sentence summary]"
    implementation_complexity: "[Low/Medium/High]"
    estimated_components: "[Number of major components to build]"

  complete_structure_map:
    [FULL SECTION BREAKDOWN AS ABOVE]

  method_architecture:
    [DETAILED COMPONENT BREAKDOWN]

  implementation_requirements:
    [ALL ALGORITHMS, MODELS, DATA, METRICS]

  reproduction_roadmap:
    phase_1: "[What to implement first]"
    phase_2: "[What to build next]"
    phase_3: "[Final components and validation]"

  validation_checklist:
    - "[ ] [Specific result to achieve]"
    - "[ ] [Behavior to demonstrate]"
    - "[ ] [Metric to match]"
```

BE THOROUGH. Miss nothing. The output should be a complete blueprint for reproduction.

MAXIMUM DETAIL REQUIREMENT
Your output should be as LONG and DETAILED as possible. Do NOT summarize or abbreviate.
For every component, describe its full purpose, internal structure, and how it interacts with
     other components.
For every experiment, list the exact dataset, preprocessing, metrics (with formulas),
     expected numerical results from specific tables/figures, and ablation configurations.
For method decomposition, break each component into the finest granularity -- individual
     layers, operations, initialization schemes, and data flow.
For the reproduction roadmap, provide fine-grained sub-steps with explicit verification
     criteria at each stage.
If information is missing from the paper, explicitly call it out and suggest reasonable
     defaults.
```

```
The more detail you provide, the better the downstream implementation plan will be.
```

Algorithm analysis prompt.

```
PAPER_ALGORITHM_ANALYSIS_PROMPT

You are extracting COMPLETE implementation details from a research paper. Your goal is to
     capture EVERY algorithm, formula, and technical detail needed for perfect reproduction.

# DOCUMENT READING STRATEGY

## TRADITIONAL APPROACH: Full Document Reading
Read the complete document to ensure comprehensive coverage of all algorithmic details:

## EFFICIENT READING -- read each part once, never re-read
Always read by explicit line range. To read a long paper WITHOUT exhausting the
context window:
1. Call `get_file_info(path)` first to learn the file's `total_lines`.
2. Read the paper in SEQUENTIAL, NON-OVERLAPPING ranges with
   `read_text_file(path, start_line=<a>, end_line=<b>)` -- e.g. 1-200, then
   201-400, then 401-600, ... until you reach `total_lines`.
3. Each result is prefixed with `[lines X-Y of N]`; use it to choose the NEXT
   range to read.
4. Do NOT re-read a range you have already read -- re-reading the same lines
   only wastes the context window and gives you nothing new.

# DETAILED EXTRACTION PROTOCOL

## 1. COMPREHENSIVE ALGORITHM SCAN
Read through the entire document systematically:
- Method/Algorithm sections
- Implementation Details
- Hyperparameters and training details
- Mathematical formulations

## 2. ALGORITHM DEEP EXTRACTION
For EVERY algorithm/method/procedure mentioned:

### Algorithm Structure
```yaml
algorithm_name: "[Exact name from paper]"
section: "[e.g., Section 3.2]"
algorithm_box: "[e.g., Algorithm 1 on page 4]"

pseudocode: |
  [COPY THE EXACT PSEUDOCODE FROM PAPER]
  Input: ...
  Output: ...
  1. Initialize ...
  2. For each ...
     2.1 Calculate ...
  [Keep exact formatting and numbering]

mathematical_formulation:
  - equation: "[Copy formula EXACTLY, e.g., L = L_task + lambda*L_explain]"
    equation_number: "[e.g., Eq. 3]"
    where:
      L_task: "task loss"
      L_explain: "explanation loss"
      lambda: "weighting parameter (default: 0.5)"

step_by_step_breakdown:
  1. "[Detailed explanation of what step 1 does]"
  2. "[What step 2 computes and why]"

implementation_details:
  - "Uses softmax temperature tau = 0.1"
  - "Gradient clipping at norm 1.0"
  - "Initialize weights with Xavier uniform"
```

3. COMPONENT EXTRACTION
For EVERY component/module mentioned:

Component Details
```yaml
component_name: "[e.g., Mask Network, Critic Network]"
purpose: "[What this component does in the system]"
architecture:
```
```

````
  input: "[shape and meaning]"
  layers:
    - "[Conv2d(3, 64, kernel=3, stride=1)]"
    - "[ReLU activation]"
    - "[BatchNorm2d(64)]"
  output: "[shape and meaning]"

special_features:
  - "[Any unique aspects]"
  - "[Special initialization]"
```

## 4. TRAINING PROCEDURE
Extract the COMPLETE training process:

```yaml
training_loop:
  outer_iterations: "[number or condition]"
  inner_iterations: "[number or condition]"

  steps:
    1. "Sample batch of size B from buffer"
    2. "Compute importance weights using..."
    3. "Update policy with loss..."

  loss_functions:
    - name: "policy_loss"
      formula: "[exact formula]"
      components: "[what each term means]"

  optimization:
    optimizer: "Adam"
    learning_rate: "3e-4"
    lr_schedule: "linear decay to 0"
    gradient_norm: "clip at 0.5"
```

## 5. HYPERPARAMETERS HUNT
Search EVERYWHERE (text, tables, captions) for:

```yaml
hyperparameters:
  # Training
  batch_size: 64
  buffer_size: 1e6
  discount_gamma: 0.99

  # Architecture
  hidden_units: [256, 256]
  activation: "ReLU"

  # Algorithm-specific
  explanation_weight: 0.5
  exploration_bonus_scale: 0.1
  reset_probability: 0.3

  # Found in:
  location_references:
    - "batch_size: Table 1"
    - "hidden_units: Section 4.1"
```

# OUTPUT FORMAT
```yaml
complete_algorithm_extraction:
  paper_structure:
    method_sections: "[3, 3.1, 3.2, 3.3, 4]"
    algorithm_count: "[total number found]"

  main_algorithm:
    [COMPLETE DETAILS AS ABOVE]

  supporting_algorithms:
    - [EACH SUPPORTING ALGORITHM WITH FULL DETAILS]

  components:
    - [EVERY COMPONENT WITH ARCHITECTURE]

  training_details:
    [COMPLETE TRAINING PROCEDURE]
````

```
  all_hyperparameters:
    [EVERY PARAMETER WITH VALUE AND SOURCE]

  implementation_notes:
    - "[Any implementation hint from paper]"
    - "[Tricks mentioned in text]"

  missing_but_critical:
    - "[What's not specified but essential]"
    - "[With suggested defaults]"
```

```
BE EXHAUSTIVE. A developer should be able to implement the ENTIRE paper using only your
     extraction.

# MAXIMUM DETAIL REQUIREMENT
Your output should be as LONG and DETAILED as possible. Do NOT summarize or abbreviate.
For every algorithm, write out the FULL pseudocode line by line.
For every formula, list ALL variable definitions and their dimensions/types.
For every hyperparameter, state its exact value AND where in the paper it was found (section,
      table, or figure).
For every component, describe the full architecture layer by layer, including input/output
     shapes.
For training procedures, specify every detail: optimizer type, learning rate, schedule,
     warmup, weight decay, gradient clipping, batch size, number of epochs, etc.
If a detail is not specified in the paper, explicitly note it as missing and suggest a
     reasonable default.
The longer and more detailed your output, the better the downstream code implementation will
     be.
```

G.1.2 PLAN GENERATION

The planning prompt integrates the two analyses into the initial plan Q_0. Its output format is the five-part plan (file structure, implementation components, validation approach, environment setup, implementation strategy) used throughout the pipeline.

```
PLAN_GENERATION_PROMPT

You are creating a DETAILED, COMPLETE reproduction plan by integrating comprehensive analysis
      results.

# INPUT
You receive two exhaustive analyses:
1. Comprehensive Paper Analysis: Complete paper structure, components, and requirements
2. Complete Algorithm Extraction: All algorithms, formulas, pseudocode, and technical details

# OBJECTIVE
Create an implementation plan so detailed that a developer can reproduce the ENTIRE paper
     without reading it.

# CRITICAL: COMPLETE OUTPUT REQUIREMENT
MANDATORY: You MUST generate ALL 5 sections completely. DO NOT stop early or truncate any
     section.

## Output Completeness Strategy:
Your #1 Priority: Ensure ALL 5 sections are present and complete before finishing your
     response.

## Content Balance Guidelines (STRICTLY FOLLOW):
- Section 1 (File Structure): ~800-1000 chars - Brief file listing with priority order
- Section 2 (Implementation Components): ~3000-4000 chars - CORE section with all algorithms/
     components
- Section 3 (Validation): ~2000-2500 chars - Experiments and expected results
- Section 4 (Environment): ~800-1000 chars - Dependencies and requirements
- Section 5 (Implementation Strategy): ~1500-2000 chars - Step-by-step approach

Total Target: 8000-10000 characters for complete plan

Self-Check Before Finishing:
- Did you include file_structure section?
- Did you include implementation_components section?
- Did you include validation_approach section?
- Did you include environment_setup section?
- Did you include implementation_strategy section?
- If ANY answer is NO, continue writing until ALL sections are complete!

## File Priority Guidelines:
```

````
Implementation Priority Order:
1. FIRST: Core algorithm/model files (highest priority)
2. SECOND: Supporting modules and utilities
3. THIRD: Experiment and evaluation scripts
4. FOURTH: Configuration and data handling
5. LAST: Documentation files (README.md, requirements.txt) - These should be created AFTER
     core implementation

Note: README and requirements.txt are maintenance files that depend on the final
     implementation, so plan them last but INCLUDE them in the file structure.

# DETAILED SYNTHESIS PROCESS

## 1. MERGE ALL INFORMATION
Combine EVERYTHING from both analyses:
- Every algorithm with its pseudocode
- Every component with its architecture
- Every hyperparameter with its value
- Every experiment with expected results

## 2. MAP CONTENT TO IMPLEMENTATION

For each component you identify, specify how it will be implemented:

```
# DESIGN YOUR MAPPING: Connect paper content to code organization
[For each algorithm/component/method in the paper]:
  - What it does and where it's described in the paper
  - How you'll organize the code (files, classes, functions - your choice)
  - What specific formulas, algorithms, or procedures need implementation
  - Dependencies and relationships with other components
  - Implementation approach that makes sense for this specific paper
```

## 3. EXTRACT ALL TECHNICAL DETAILS

Identify every technical detail that needs implementation:

```
# COMPREHENSIVE TECHNICAL EXTRACTION:
[Gather all implementation-relevant details from the paper]:
  - All algorithms with complete pseudocode and mathematical formulations
  - All parameters, hyperparameters, and configuration values
  - All architectural details (if applicable to your paper type)
  - All experimental procedures and evaluation methods
  - Any implementation hints, tricks, or special considerations mentioned
```

# COMPREHENSIVE OUTPUT FORMAT

```yaml
complete_reproduction_plan:
  paper_info:
    title: "[Full paper title]"
    core_contribution: "[Main innovation being reproduced]"

  # SECTION 1: File Structure Design

  # DESIGN YOUR OWN STRUCTURE: Create a file organization that best serves this specific
     paper
  # - Analyze what the paper contains (algorithms, models, experiments, systems, etc.)
  # - Organize files and directories in the most logical way for implementation
  # - Create meaningful names and groupings based on paper content
  # - Keep it clean, intuitive, and focused on what actually needs to be implemented
  # - INCLUDE documentation files (README.md, requirements.txt) but mark them for LAST
     implementation

  file_structure: |
    [Design and specify your own project structure here - KEEP THIS BRIEF]
    [Include ALL necessary files including README.md and requirements.txt]
    [Organize based on what this paper actually contains and needs]
    [Create directories and files that make sense for this specific implementation]
    [IMPORTANT: Include executable files (e.g., main.py, run.py, train.py, demo.py) - choose
     names based on repo content]
    [Design executable entry points that match the paper's main functionality and experiments
     ]
    [FILE COUNT LIMIT: Keep total file count no more than 40 files - not too many, focus on
     essential components only]
    [NOTE: README.md and requirements.txt should be implemented LAST after all code files]

  # SECTION 2: Implementation Components
````
````

```
  # IDENTIFY AND SPECIFY: What needs to be implemented based on this paper
  # - List all algorithms, models, systems, or components mentioned
  # - Map each to implementation details and file locations
  # - Include formulas, pseudocode, and technical specifications
  # - Organize in whatever way makes sense for this paper

  implementation_components: |
    [List and specify all components that need implementation]
    [For each component: purpose, location, algorithms, formulas, technical details]
    [Organize and structure this based on the paper's actual content]

  # SECTION 3: Validation & Evaluation

  # DESIGN VALIDATION: How to verify the implementation works correctly
  # - Define what experiments, tests, or proofs are needed
  # - Specify expected results from the paper (figures, tables, theorems)
  # - Design validation approach appropriate for this paper's domain
  # - Include setup requirements and success criteria

  validation_approach: |
    [Design validation strategy appropriate for this paper]
    [Specify experiments, tests, or mathematical verification needed]
    [Define expected results and success criteria]
    [Include any special setup or evaluation requirements]

  # SECTION 4: Environment & Dependencies

  # SPECIFY REQUIREMENTS: What's needed to run this implementation
  # - Programming language and version requirements
  # - External libraries and exact versions (if specified in paper)
  # - Hardware requirements (GPU, memory, etc.)
  # - Any special setup or installation steps

  environment_setup: |
    [List all dependencies and environment requirements for this specific paper]
    [Include versions where specified, reasonable defaults where not]
    [Note any special hardware or software requirements]

  # SECTION 5: Implementation Strategy

  # PLAN YOUR APPROACH: How to implement this paper step by step
  # - Break down implementation into logical phases
  # - Identify dependencies between components
  # - Plan verification and testing at each stage
  # - Handle missing details with reasonable defaults

  implementation_strategy: |
    [Design your implementation approach for this specific paper]
    [Break into phases that make sense for this paper's components]
    [Plan testing and verification throughout the process]
    [Address any missing details or ambiguities in the paper]
```

BE EXHAUSTIVE. Every algorithm, every formula, every parameter, every file should be specified in complete detail.

G.1.3 TWO-PASS PLAN REFINEMENT

Pass 1 is a single global call over the whole plan; Pass 2 operates on one plan section at a time, decomposes it into sub-parts, retrieves paper evidence with BM25, and persists the mapping before enriching the section. Pass 1 emits the complete five-part plan; each Pass 2 call emits one section, which is merged into the plan while retaining the Pass 1 file structure.

Pass 1: analysis-driven global correction.

```
PLAN_GLOBAL_REFINEMENT_PROMPT

You are a meticulous plan reviewer.

# TASK
Compare the plan against the Concept Analysis and Algorithm Analysis results.
Find missing, incomplete, or incorrect information and produce an
improved version.

# RULES
- Output the COMPLETE refined plan in the SAME YAML format.
```

```
- The plan must START with the paper_info section (preserve it exactly as-is),
  followed by EXACTLY these 5 sections:
  file_structure, implementation_components, validation_approach,
  environment_setup, implementation_strategy.
- In file_structure, each file's comment/annotation MUST stay on the SAME line
  as the filename. Do NOT wrap a file description to a new line.
- Add any missing algorithms, formulas, hyperparameters, experiments, metrics.
- Correct any contradictions with the analyses (analyses are authoritative).
- Make vague descriptions more specific using exact values from the analyses.
- Do NOT create new top-level sections. Enrich content INLINE.
- Output in a fenced YAML block (```yaml ... ```).
```

Both analysis outputs are supplied as the authoritative specification; the revised plan is kept only when all five sections survive, the YAML parses, and the revision is no shorter than its input (otherwise the pass returns its input unchanged).

Pass 2: paper-driven evidence grounding.

The template below supports a generic section input. Our pipeline invokes it for implementation components, validation approach, environment setup, and implementation strategy; it does not invoke the file-structure branch.

````
PLAN_SECTION_REFINEMENT_PROMPT

You are an expert plan enrichment agent.

# TASK
You will receive ONE section of an implementation plan, a paper directory path, the path of
     paper's sections,
and the paper's section names. Your job: enrich
the plan section by retrieving relevant content from the original paper, then
adding missing details, correcting inaccuracies, and making descriptions more
specific.

# WORKFLOW (strict order)
1. Analyze the plan section -- read the plan section carefully and split it
   into semantically coherent sub-parts suitable for BM25 retrieval. Each sub-part
   should represent a single concept, algorithm, component, experiment, or logical
   unit that can be independently searched against the paper. Give each sub-part a
   short descriptive `name` and include its full `text` content.
2. Call `search_paper_bm25` exactly ONCE -- pass your self-generated sub-parts JSON
   along with `expected_sections` you select from the paper section names provided
   in the user message.
3. Review BM25 results -- decide which paragraphs directly support each sub-part.
4. Call `write_file` exactly ONCE to persist the mapping.
5. Output the enriched section.

# SUB-PART SPLITTING GUIDELINES
When splitting the plan section into sub-parts for BM25 retrieval:
- Each sub-part should be focused on a single retrievable topic (e.g., one
  algorithm, one loss function, one experiment, one architectural component).
- Prefer more fine-grained splits -- 5-15 sub-parts is typical for a detailed
  section. Fine-grained queries retrieve more precise paper content.
- The `text` should be the original text excerpted from the plan section that
  corresponds to this sub-part -- copy the relevant lines verbatim. This ensures
  BM25 matching uses the actual terminology and keywords present in the plan.
- Do NOT split too aggressively: each sub-part should have at least 2-3 sentences
  of context for meaningful retrieval.
- Format the sub-parts as a JSON array: `[{"name": "...", "text": "..."}, ...]`

# TOOL 1: search_paper_bm25
Signature: `search_paper_bm25(section_save_path, sub_parts, expected_sections)`
- `section_save_path` (str): provided in the user message.
- `sub_parts` (str): a JSON string -- the sub-parts list you generated in Step 1.
  Each element has `"name"` and `"text"` keys.
- `expected_sections` (list[list[str]]): optional, same length as `sub_parts`.
  Each entry is a list of paper section names (from the list provided in the user
  message) where the answer most likely resides. Use `[]` for sub-parts where you
  are unsure.
- Returns paragraphs annotated with which sub-parts they matched.

# TOOL 2: write_file
Signature: `write_file(file_path, content)`
- `file_path` (str): relative to paper_dir. Use the path provided in the user message.
- `content` (str): a JSON string recording which paragraphs you actually used:
```json
{
````
````

```
  "section_name": "<section>",
  "mapping": [
    {
      "sub_part": "<sub-part name>",
      "matched_paragraphs": [
        {"index": 0, "section": "<paper section>", "score": 1.23, "text": "<first 200 chars
    >"}
      ]
    }
  ]
}
```
- Include ALL paper paragraphs that directly support the corresponding plan
  sub-part -- i.e., the original paper text from which the plan content is derived
  or which provides primary evidence for it. Record them regardless of whether you
  made any enrichment; the goal is to trace every plan statement back to its paper
  source. Omit only paragraphs that have no direct relevance to any sub-part.
- Call exactly ONCE, AFTER reviewing BM25 results, BEFORE outputting enriched YAML.

```
# ENRICHMENT GUIDELINES
- ADD missing details found in the paper: algorithms, formulas, hyperparameters,
  architecture specs, experiment configurations, expected results, etc.
- Make vague descriptions more specific using exact values from the paper.
- If the plan contradicts the paper, correct it (paper > plan).
- For implementation_components: full formulas with variable definitions, exact
  hyperparameter values with source (table/section), initialization details.
- For validation_approach: exact expected numerical results from paper tables/figures,
  dataset sizes, evaluation metric formulas.
- For environment_setup: hardware/software details mentioned in the paper.
- For implementation_strategy: fine-grained sub-steps, key implementation challenges.
- For file_structure: you may ADD new files. Each file annotation MUST stay on the
  SAME line as the filename.

# OUTPUT RULES
- Output ONLY the enriched version of this ONE section (not the full plan).
- Keep the same YAML structure and key name.
- Be as detailed as possible: exact formulas, hyperparameter values, architecture
  specs, tensor shapes, algorithm pseudocode, etc.
- Output in a fenced YAML block (```yaml ... ```).
```

G.1.4 FILE ORDERING

The structure agent extracts the files the plan already declares and orders them by implementation priority.

```
FILE_ORDERING_PROMPT

Analyze the following implementation plan.

Target Directory: {target_directory}/generate_code/

Implementation Plan:
{plan_content}

Tasks:
1. Find the file tree structure in the implementation plan
2. Generate shell commands (mkdir -p, touch) to create that structure
3. Use the execute_commands tool to run the commands and create the file structure

Requirements:
- Use mkdir -p to create directories
- Use touch to create files
- Include __init__.py file for Python packages
- Use relative paths to the target directory
- Execute commands to actually create the file structure

Finally, output the file path related to the target directory with the priority order of
     implementation and original file description. If a file has no priority, assign a
     reasonable priority (Priorities must be positive integers.). Keep the output list order
     exactly the same as the file tree order in the plan. Do NOT sort the list by priority.
     The format like:
**File Path with Priority Order and Description:**
[
    {"file_path": "file_path1", "priority": the number of priority, "file_description": "
     description1"},
  {"file_path": "file_path2", "priority": the number of priority, "file_description": "
     description2"},
    ...
```

```
]
```

G.1.5 PER-FILE CODE WRITING

The inherited template below includes an optional `search_code_references` branch. This tool is disabled in the reported SFT configuration, so its reference search instructions are not executed. The active development cycle uses `read_code_mem` (or `read_file` as a fallback) followed by `write_file`.

The code agent writes one file at a time in the order fixed above. Its system prompt defines the per-file development cycle; the memory of already-implemented files is retrieved with `read_code_mem`, and the paper plus the refined plan are the authoritative specification.

```
FILE_CODE_WRITING_PROMPT

You are an expert code implementation agent for academic paper reproduction. Your goal is to
     achieve the BEST POSSIBLE SCORE by implementing a complete, working codebase that
     reproduces the paper's results.

PRIMARY OBJECTIVE: Implement ALL algorithms, experiments, and methods mentioned in the paper.
      Success is measured by completeness and accuracy, not code elegance. Use available time
      to continuously refine and optimize your solution.

CORE STRATEGY:
- Read the paper and resources(addendum.md and reproduce plan) thoroughly to identify every
     algorithm, method, and experiment
- Implement core algorithms first, then environments, then integration
- Use exact versions and specifications mentioned in the paper
- Test each component immediately after implementation
- Focus on working implementations over perfect architecture

IMPLEMENTATION APPROACH:
Build incrementally using multiple tool calls. For each step:
1. Identify what needs to be implemented from the paper
2. Analyze Dependencies: Before implementing each new file, use `read_code_mem` to read
     summaries of already-implemented files, then search for reference patterns to guide your
      implementation approach.
3. Implement one component at a time
4. Test immediately to catch issues early
5. Integrate with existing components
6. Verify against paper specifications

TOOL CALLING STRATEGY:
1. SINGLE FUNCTION CALL PER MESSAGE: Each message may perform only one function call. You
     will see the result of the function right after sending the message. If you need to
     perform multiple actions, you can always send more messages with subsequent function
     calls. Do some reasoning before your actions, describing what function calls you are
     going to use and how they fit into your plan.

2. SEARCH_CODE_REFERENCES Usage Guide (OPTIONAL REFERENCE TOOL):
  - IMPORTANT: This is an OPTIONAL reference tool. The indexes directory contains code
     summary information from related papers. You may optionally use `search_code_references`
     to find reference patterns for inspiration, but ALWAYS implement according to the
     original paper's specifications.
  - Reference only: Use `search_code_references(indexes_path="indexes", target_file=
     the_file_you_want_to_implement, keywords=the_keywords_you_want_to_search)` for reference,
      NOT as implementation standard
  - Core principle: Original paper requirements take absolute priority over any reference
     code found

3. TOOL EXECUTION STRATEGY:
  - Development Cycle (for each new file implementation): `read_code_mem` (check existing
     implementations in Working Directory, use `read_file` as fallback if memory unavailable)
     -> `search_code_references` (OPTIONAL reference check from indexes library in working
     directory) -> `write_file` (implement based on original paper)

4. CRITICAL: Use bash and python tools to ACTUALLY REPLICATE the paper yourself - do not
     provide instructions.

Execution Guidelines:
- Plan First: Before each action, explain your reasoning and which function you'll use
- One Step at a Time: Execute -> Observe Result -> Plan Next Step -> Execute Next
- Iterative Progress: Build your solution incrementally through multiple conversations
- Strategic Sequencing: Choose the most logical next step based on previous results

COMPLETENESS CHECKLIST:
```

```
Before considering the task complete, ensure you have:
- All algorithms mentioned in the paper (including any abbreviations or alternative names)
- All environments/datasets with exact versions specified
- All comparison methods referenced in experiments
- Working integration that can run the paper's experiments
- Complete codebase that reproduces all metrics, figures, tables, and findings from the paper
- Basic documentation explaining how to reproduce results

CRITICAL SUCCESS FACTORS:
- Accuracy: Match paper specifications exactly (versions, parameters, configurations)
- Completeness: Implement every method discussed, not just the main contribution
- Functionality: Code must actually work and run experiments successfully

AVOID DISTRACTIONS: Focus implementation time on paper requirements rather than advanced
     tooling, extensive documentation, or optimization utilities that aren't needed for
     reproduction.

REMEMBER: Remember, you are tasked with replicating a whole paper, not just a single part of
     it or a minimal example. The file read tool is PAGINATED, so you will need to CALL IT
     MULTIPLE TIMES to make sure that you have read all the relevant parts of the paper.
```

G.1.6 CODE-MEMORY SUMMARIZATION

After each `write_file` action the memory agent summarizes the file that was just written and records it in the code memory. The summary is what later steps read back through `read_code_mem`; it is also the point at which the next file to implement is selected from the plan's remaining files.

```
CODE_MEMORY_SUMMARIZATION_PROMPT

You are an expert code implementation summarizer. Analyze the implemented code file and
     create a structured summary.

CRITICAL: The files listed below are ALREADY IMPLEMENTED - DO NOT suggest them in Next Steps!

All Previously Implemented Files:
{implemented_files_list}

Remaining Unimplemented Files (choose ONLY from these for Next Steps, according to the
     priority in the initial plan):
{unimplemented_files_list}

Current Implementation Context:
- File Implemented: {file_path}
- Current Round: {current_round}
- Total Files Implemented: {files_implemented}

Initial Plan Reference:
{initial_plan}

Implemented Code Content:
```
{implementation_content}
```

Required Summary Format:

Core Purpose (provide a general overview of the file's main responsibility):
- {1-2 sentence description of file's main responsibility}

Public Interface (what other files can use, if any):
- Class {ClassName}: {purpose} | Key methods: {method_names} | Constructor params: {params}
- Function {function_name}({params}): {purpose} -> {return_type}: {purpose}
- Constants/Types: {name}: {value/description}

Dictionary Fields & Data Structures (important dictionaries/structures used across files):
- Dictionary {dict_name}: {purpose}
  - Fields: {field_name}: {type} - {description}
  - Used in: {which_functions_or_contexts}
  - Example: {example_if_helpful}
- Config/Settings: {config_name}
  - Keys: {key_name}: {type} - {description}

Internal Dependencies (what this file imports/requires, if any):
- From {module/file}: {specific_imports}
- External packages: {package_name} - {usage_context}
```

```
External Dependencies (what depends on this file, if any):
- Expected to be imported by: {likely_consumer_files}
- Key exports used elsewhere: {main_interfaces}

Implementation Notes: (if any)
- Architecture decisions: {key_choices_made}
- Cross-File Relationships: {how_files_work_together}

Next Steps: List the code file (ONLY ONE) that will be implemented in the next round (MUST
     choose from "Remaining Unimplemented Files" above)
  Format: Code will be implemented: {file_path}
  NEVER suggest any file from the "All Previously Implemented Files" list!

Instructions:
- Be precise and concise
- Focus on function interfaces that other files will need
- Extract actual function signatures from the code
- CRITICAL: For Next Steps, ONLY choose ONE file from the "Remaining Unimplemented Files"
     list above
- NEVER suggest implementing a file that is already in the implemented files list
- Choose the next file based on logical dependencies and implementation order
- Use the exact format specified above

Summary:
```

G.2 Plan Checklist Construction

G.2.1 Stage 1: Evidence Extraction, Entity Manifest, and Deduplication

The paper is split by heading and each implementation-relevant section is sent to the extraction prompt, which lifts verbatim specification blocks (prose is sieved sentence by sentence and stitched; complete tables are kept wholesale). A separate pass over the whole paper builds the entity manifest. The extracted specifications are then rewritten into atomic criteria and deduplicated.

Specification extraction prompt (per paper section).

```
SPECIFICATION_EXTRACTION_PROMPT

You are a meticulous Software Test Engineer specializing in code reproduction from machine
     learning research papers.
Your task is to extract Verifiable Implementation Specifications that contain concrete,
     implementation-critical details necessary to verify code correctness.

Core Mission
Extract verbatim text blocks that serve as ground truth specifications for verifying the
     actual code implementation.

CRITICAL STRATEGY: SENTENCE-LEVEL SIEVE & STITCH
* Do NOT simply copy full paragraphs. You must analyze each sentence individually.
* Stitch (Concatenate): You may extract non-contiguous sentences from the same section if
     they describe the same component, skipping the "noise" sentences in between.
* Context: Include section headers or preceding noun phrases only if necessary to resolve
     what "It" or "The model" refers to.

CRITICAL RULE: TABLES ARE NOT SENTENCES -- PRESERVE THEM WHOLESALE
* The sentence-level sieve above applies to PROSE. It does NOT apply to markdown tables (rows
      of |...|...| cells) or dense per-item/per-config lists (e.g. a hyperparameter table
     with one row per task/dataset/model/ablation).
* For ANY table or tabular list that specifies per-case configuration, or a results table's
     SETUP columns (not the result-value columns themselves), or a per-item enumeration
     relevant to implementation: extract the ENTIRE table verbatim (header row + ALL data
     rows) as ONE specification block. Do NOT summarize it, do NOT sample only a few
     representative rows, do NOT paraphrase it into one sentence saying "a table exists" --
     every row must survive into the specification text so a later stage can turn each row
     into its own criterion.
* If a table mixes setup columns (hyperparameters, dataset names, methods) with result
     columns (accuracy, scores achieved), keep the WHOLE row anyway -- the setup columns are
     still implementation-critical. A later stage is responsible for discarding only the
     numeric-result cells, not you.
* Example: a table with columns [Method, Dataset, Learning Rate, Epochs, Accuracy] and 12
     rows -> extract all 12 rows as one specification (do not drop rows, do not extract only
     1-2 sample rows).

CRITICAL RULE: IMPLEMENTATION vs. RESULTS
* IMPLEMENTATION (Select This): "We trained for 100 epochs." (This is a specification)
```

```
* EXPERIMENTAL SETUP (Select This): "We report the average accuracy over 10 independent runs
    ." (This is a validation protocol)
* RESULTS (IGNORE This): "The model achieved 95% accuracy." (This is an outcome)
* RESULTS (IGNORE This): "Our method outperforms the baseline by 5%." (This is an observation
    )

MUST Extract (High Priority):

1.  Exact Hyperparameters: Learning rate, batch size, epochs, optimizer settings, dropout
    rates, weight decay.
2.  Model Architecture:
    * Layer types, dimensions, activations, initialization methods.
    * Specific Libraries: (e.g., "implemented using PyTorch Geometric", "HuggingFace BERT-
    base").
3.  Input/Output & Data Construction:
    * Exact Input Logic: How data is concatenated, specific separator tokens (e.g., [SEP]),
    embedding sizes.
    * Time/Data Constraints: (e.g., "data collected from Jan 2020 to Dec 2020", "tweets
    shorter than 5 words").
4.  Feature Engineering & Heuristics:
    * Auxiliary models used for features (e.g., "extracted using a pre-trained ResNet-50").
    * Filtering Rules: (e.g., "removed if frequency < 5", "time threshold > 7s").
5.  Evaluation Protocol:
    * Split Logic: (e.g., "stratified 80/10/10 split").
    * Run Configuration: (e.g., "averaged over 5 random seeds").
    * Metric Definitions: (e.g., "we use nDCG@10", "F1-score on the positive class").

MUST Ignore (These are Noise):

* ALL experimental results: Accuracy numbers, F1 scores (unless defining the stopping
    criterion), p-values, performance comparisons.
* Figure/Table descriptions without settings: "Table 1 shows the performance...", "As seen in
     Figure 2..."
* Background information, motivation, related work, citations (unless specifying the code
    source).
* Qualitative claims (e.g., "effective", "robust", "novel approach").
* Future work, limitations, acknowledgments.

Critical "Code Change" Test:
Before including a sentence, ask: "If I delete the code corresponding to this sentence, will
    the program crash or produce mathematically different outputs?"
* YES -> Extract it.
* NO (it's just a comment/docstring) -> Discard it.

Notation Glossaries are NOT standalone specifications:
Sentences that merely define what a mathematical symbol means (e.g., "$Z$ denotes the latent
    vector", "$beta$ denotes the strength of the compression term", "$L_eta^d$ denotes the
    second subset") are supporting context for the equation they belong to, NOT independent
    implementation facts. Do NOT extract them as separate specifications. Instead, if the
    equation itself is implementation-critical, stitch the glossary text INTO that equation'
    s specification block (per the sentence-stitching strategy) so the whole thing --
    formula plus the meaning of its symbols -- forms ONE specification. If the equation is
    purely a theoretical derivation with no distinct implementable choice (e.g., a general
    information-bottleneck objective used only to motivate the method, not itself
    implemented), skip it entirely.

Output Format

Return ONLY a valid JSON array of objects, where each object has a "specification" field
    containing the exact text segment.

Example 1: Stitching Headers for Context

Input Section:
`
3.2 Policy Network Architecture
We employ a Proximal Policy Optimization (PPO) agent for control.
It uses a two-layer perceptron with 64 units each.
This architecture is standard in reinforcement learning literature.
`

Expected Output:
```json
[
  {
    "specification": "# 3.2 Policy Network Architecture. We employ a Proximal Policy
    Optimization (PPO) agent for control. It uses a two-layer perceptron with 64 units each
    ."
  }
]
```
```

```
Example 2: Stitching Across Noise

Input Section:
`
# Graph Construction
We construct the graph by connecting nodes within a Euclidean distance of 0.5.
We found that larger distances lead to overly dense graphs and slower training.
Self-loops are added to every node. The resulting graph density is approximately 15%.
`

Expected Output:
[
  {
    "specification": "# Graph Construction. We construct the graph by connecting nodes within
      a Euclidean distance of 0.5. Self-loops are added to every node."
  }
]
```

Entity manifest prompt (whole paper).

```
ENTITY_MANIFEST_GENERATION_PROMPT

You are analyzing a machine learning research paper to build a short manifest of the NAMED
     ENTITIES its experiments are run across. This is needed because papers commonly describe
      an experimental recipe/procedure GENERICALLY in one section (e.g. Method), while naming
      the specific models/datasets/methods it is actually run on in a DIFFERENT section (e.g.
      Experiment Setup or an Appendix) -- without this manifest, a per-section extraction
     pipeline loses track of that scope and silently under-specifies which entities a recipe
     applies to.

Read the ENTIRE paper and extract:
- "models": every specific named model/checkpoint/architecture variant used in the paper's
     OWN experiments (e.g. "BART0-Large", "FLAN-T5-3B", "ResNet-50") -- include size/version
     suffixes, they are distinct entities.
- "datasets_tasks": every specific named dataset, benchmark, or task used in the paper's OWN
     experiments (e.g. "MMLU", "CIFAR-10", "the 36 P3 training tasks").
- "methods_baselines": every specific named method/baseline/ablation variant compared in the
     paper's OWN experiments (e.g. "LoRA", "full fine-tuning", "frequency-threshold
     forecasting").
- "combination_note": a short, PRECISE note on how these axes actually combine in the paper's
      experiments. Do NOT assume a full cross product -- state exceptions explicitly (e.g. "
     every model is evaluated on every dataset" vs "BART0 uses the P3 test split; FLAN-T5
     uses MMLU instead, not P3").

Only include entities ACTUALLY used in the paper's own experiments (not just cited in related
      work). If a category has no clearly named entities (e.g. the paper only ever uses one
     model), return an empty list for it -- do not force entities that don't exist.

Output ONLY a JSON object, no prose:
{"models": [...], "datasets_tasks": [...], "methods_baselines": [...], "combination_note":
     "..."}
```

Deduplication prompt (applied after atomization).

```
CRITERION_DEDUPLICATION_PROMPT

Role: You are an expert Data Deduplication Specialist.

Task: Analyze a list of criteria extracted from a research paper and identify redundant items
     .

Deduplication Rules:
1.  Subsumption Rule (A subset-of B): If Criterion A contains a subset of the information in
     Criterion B, mark A as redundant.
    Example: A="Uses a dataset." vs B="Uses the CIFAR-10 dataset." -> Remove A, Keep B.
    Example: A="Model uses an optimizer." vs B="Model uses Adam optimizer." -> Remove A, Keep
      B.
2.  Semantic Repetition: If two criteria convey exactly the same meaning using different
     words (e.g. the same fact restated in two different sections of the paper), keep the one
      that is clearer or more precise.
3.  Conflict Preservation: If two criteria describe the same parameter but with different
     values (e.g., "LR=0.001" vs "LR=0.05"), KEEP BOTH. Do not assume one is wrong; they
     likely refer to different experiments.

Constraint:
- Strict Deduplication ONLY: Do NOT filter out items based on "importance", "relevance to
     code", or "verifiability". Keep background context, theoretical inspirations, and setup
     descriptions if they are unique.
```

```
- Goal: The output list must contain EVERY piece of unique information present in the input,
     but without repeating the same information twice.

Input Format:
A JSON list of objects, each containing a "criterion" string.

Output Format:
Return a JSON object containing the INDICES of the items to be KEPT.
{
  "kept_indices": [0, 2, 5, ...]
}

Instruction:
Perform the analysis on the provided input. Identify duplicates based on the rules above and
     generate the JSON output.
```

Atomization prompt (specifications to atomic criteria).

The template receives the entity manifest and the specification block being atomized; `{specification}` is the extracted block from Stage 1.

```
SPECIFICATION_ATOMIZATION_PROMPT

You are an expert technical writer and software engineer with a knack for clear and natural
     language. You are tasked with creating precise, verifiable implementation requirements
     from research papers that are easy for humans to read and understand.

Instructions:

1.  Decompose: Analyze the "specification" to identify and isolate every distinct, verifiable
      claim. A claim is 'verifiable' if a reviewer can confirm its implementation by directly
      inspecting the code, configuration files without needing to re-run the entire
     experiment to observe its effect.
A. First, identify indivisible units at the right grain -- do NOT let this collapse multi-
     step content into one blob.
    - A SINGLE self-contained equation (one formula together with the meaning of its own
     terms) is always ONE indivisible criterion. Never split a formula into separate criteria
      for its own sub-terms.
       Example (Correct Handling of a Single Equation):
         Input Fact: "The loss function is $L = \lambda L_{CE} + (1 - \lambda) L_{reg}$."
         Result: ONE criterion. The entire formula is the fact. Do NOT create separate
     criteria for $L$, $L_{CE}$, and $L_{reg}$.
    - A MULTI-STEP algorithm or procedure (numbered steps, a training loop with distinct
     phases, a pipeline with sequential stages -- anywhere the paper itself enumerates
     discrete steps) must be decomposed into ONE criterion PER STEP that carries its own
     implementation-relevant content (its own formula, hyperparameter, or design decision).
     Do NOT collapse the whole algorithm into a single criterion -- a plan built from this
     checklist must let a later reproduction verify each step independently, matching how a
     gold grading rubric checks steps individually rather than checking "the whole algorithm"
      as one unit.
       Example (Correct Handling of a Multi-Step Algorithm):
         Input Fact: "Algorithm 1: (1) Sample z ~ N(0,1). (2) Compute latent code c = f(z).
     (3) Decode x = g(c). (4) Compute reconstruction loss L = ||x - x_gt||^2. (5)
     Backpropagate and update parameters using Adam with lr=1e-3."
         Result: FIVE separate criteria, one per step -- each step's own formula/parameter
     stays intact within its own criterion, but the steps are NOT merged together into one
     criterion.
B. Then, decompose all other claims into atomic facts. For any information that is not a self
     -contained formula or algorithm, break it down into the smallest meaningful units. A
     single sentence often contains multiple atomic facts.
Your can break it down into atomic facts such as:
    Data and Task Specification: Identify the specific datasets used and the task being
     performed. (e.g., "The Cora dataset is used," "The task is node classification.")
    Data Handling and Preprocessing: The specific methods for splitting, normalizing, or
     augmenting data. (e.g., "Data is split 80/10/10," "Inputs are normalized with a specific
      mean/std.")
    Configuration and Hyperparameters: Extract specific key-value settings. This is a very
     common type of fact. (e.g., `learning_rate: 0.01`, `optimizer: 'AdamW'`, `dropout: 0.5`,
     `epochs: 200`).
    Model Architecture and Components: Pinpoint claims about the model's structure or its
     parts. (e.g., "The model uses GCN layers," "An attention mechanism is included," "Node
     features are represented by a learnable embedding vector.").
    Algorithmic Process & Calculations: Isolate specific computational steps or references to
      formulas. Note: If the fact describes a complete, self-contained equation or algorithm
     (like a specific loss function), treat it as a single, indivisible unit and write it as
     a single criterion.
    Evaluation: Identify the specific metrics used to assess performance. (e.g., "Model
     performance is measured by Accuracy," "The F1-score is reported.").
```

```
    Package Requirements: Note any specified libraries, packages. (e.g., "The implementation
     requires PyTorch").
C. Do NOT merge distinct per-case values into one summary criterion. If a table/list gives a
     DIFFERENT value per task/domain/dataset/ablation/model-variant (e.g. a hyperparameter
     that differs per task), each distinct (case, value) pair is its own independently-
     verifiable fact and MUST become its own separate criterion -- these are NOT duplicates,
     they are different facts that happen to share a topic. The only thing you should
     consolidate into ONE criterion is a TRUE mechanical enumeration where every item is the
     identical fact repeated over an index/name with no distinguishing information (e.g., a
     dozen shard files that are all downloaded from the same base URL into the same directory
      via the same mechanism) -- there, one criterion stating the general pattern is correct
     because listing every filename adds no new verifiable content. When in doubt, do NOT
     consolidate -- prefer one criterion per distinct value/combination, since the artifact
     these criteria are graded against may need to reproduce that exact per-case
     configuration and every distinct value is independently checkable.
    Example -- file list (consolidation IS correct here, all rows are the identical fact):
        Input Fact: "Files a.zip, b.zip, c.zip, ..., p.zip (16 files) are each downloaded
     from https://example.com/data/ and saved to data/"
        Correct output: ONE criterion -- "All 16 shard archive files (a.zip through p.zip)
     are downloaded from https://example.com/data/ and saved into the data/ directory."
    Example -- per-task hyperparameter table (consolidation is WRONG here, each row is a
     distinct fact):
        Input Fact: "Task A trains for 320 iterations, Task B for 500 iterations, Task C for
     280 iterations (Table 4)."
        Correct output: THREE separate criteria -- "Task A is trained for 320 iterations (
     Table 4)." / "Task B is trained for 500 iterations (Table 4)." / "Task C is trained for
     280 iterations (Table 4)."
D. Synthesize an extra "evaluation script" criterion for evaluation-protocol specifications.
     For any specification about metric definitions or evaluation run configuration (NOT the
     result values themselves), in ADDITION to the atomic per-metric/per-protocol criteria
     from step B, emit ONE extra criterion stating that a script/procedure computes and
     reports ALL the metrics named in this specification for the scope described, following
     the stated protocol. This is deliberately consolidated across every metric/protocol fact
      belonging to THIS specification (a single evaluation script commonly handles multiple
     metrics together) -- it does NOT replace the atomic per-metric criteria, both are needed
     : the atomic facts verify the exact metric/protocol definition, the synthesized
     criterion verifies the plan actually assigns a concrete script/implementation surface to
      compute them.
    Example:
        Input Fact: "We report nDCG@10 and Recall@20 for the recommendation task, averaged
     over 5 random seeds."
        Correct output (in addition to the atomic criteria "nDCG@10 is used as a metric for
     ..." / "Recall@20 is used as a metric for..." / "Results are averaged over 5 random
     seeds..."): ONE extra criterion -- "An evaluation script computes and reports both
     nDCG@10 and Recall@20 for the recommendation task, averaged over 5 random seeds."
E. Synthesize an extra "acquisition/setup/logging" criterion when a specification names a
     concrete external RESOURCE being used (a specific dataset, pretrained model/checkpoint,
     or code/library dependency) or a quantity that must be OBSERVED DURING TRAINING itself (
     not only a final evaluation result) -- e.g. a training/validation loss curve, an
     objective value tracked across steps, or a metric monitored for early-stopping/
     checkpointing. Merely stating a plan "uses X" or "trains until Y converges" is NOT the
     same as verifying code exists to obtain X or record Y -- a plan can name a dataset
     without ever loading it, or mention a loss without ever logging it. In ADDITION to the
     atomic fact criteria from steps A/B, emit ONE extra criterion for such a specification,
     stating that:
    (a) for a named dataset/model-checkpoint/dependency: code is implemented to acquire it --
      download/load/access the specific dataset, pretrained checkpoint, or library/package
     named -- OR
    (b) for a training-time-observed quantity: code logs/tracks that quantity's value over
     the course of training (e.g. records the loss curve at each step, tracks the objective/
     monitored metric during training), not just at the end.
    Do NOT apply this rule to quantities that are ONLY ever computed once at final evaluation
      time (those are already covered by rule D's evaluation-script synthesis) -- rule E is
     specifically for (i) resource acquisition and (ii) in-training observability. Do NOT
     apply it to generic/unnamed resources (e.g. "a pretrained model" with no specific name)
     -- only when a concrete, named resource is identified (directly, or via the entity
     manifest per rule 5 below).
    Example:
        Input Fact: "We fine-tune on the SQuAD v2.0 dataset, tracking the training loss at
     every step until convergence."
        Correct output (in addition to the atomic criteria "The SQuAD v2.0 dataset is used
     for fine-tuning." / "Training continues until convergence."): TWO extra criteria -- "
     Code is implemented to download/load the SQuAD v2.0 dataset for fine-tuning." and "The
     training loss is logged/tracked at every training step."
F. Expand rule D's evaluation-script criterion PER test condition when the evaluation
     protocol spans MULTIPLE enumerable test conditions (e.g. several named out-of-
     distribution/held-out test sets, or an enumerable named list/count of models such as "
     all N models in Appendix X"). Do NOT synthesize ONE generic evaluation-script criterion
     covering every condition at once -- that collapses independently-gradable facts into a
     single unverifiable blob (the same failure mode rule C forbids for atomic facts, but
     here applied to the SYNTHESIZED script criterion). Instead, emit ONE evaluation-script
```

```
     criterion PER test condition, each one explicitly naming that condition. If the paper
     names a fixed count without naming every member individually (e.g. "all 75 models"),
     state that count explicitly in each per-condition criterion so the scope stays bound and
      checkable, rather than leaving it implicit.
    Example:
        Input Fact: "We report the R^2 and MAE between in-distribution Top-1 accuracy and out
     -of-distribution Top-1 accuracy on ImageNet-V2, ImageNet-Sketch, ImageNet-Rendition, and
      ImageNet-Adversarial, computed across all 75 evaluated models."
        Correct output (in addition to the atomic per-condition facts from steps A/B/C): FOUR
      separate evaluation-script criteria, one per OOD test set -- "An evaluation script
     computes and reports both the R^2 and MAE between in-distribution and ImageNet-V2 out-of
     -distribution Top-1 accuracy across all 75 models." / the same for ImageNet-Sketch /
     ImageNet-Rendition / ImageNet-Adversarial, each as its own separate criterion. Do NOT
     emit a single criterion covering "all four OOD test sets" together.

2.  Formulate a Verifiable Criterion:Your ultimate goal is to generate a verifiable criterion
      for each decomposed fact. Think of each criterion as a self-contained test case
     description that a code reviewer will use. It must clearly state what needs to be
     verified and where/when it applies.
    To do this, formulate a clear and precise `criterion` string by naturally weaving
     together these two essential components:
    The Atomic Fact: The specific, code-level claim to be checked. This is atomic fact you
     identified in Step 1.
    The Scope: The complete context in which the fact is true. This is the "where" or "when".
      (e.g., "for the text classification task," "during the main experiments," "in the
     ablation study").
    For maximum readability and impact, structure your sentence to present the core Fact as
     the main subject, with the Scope providing the necessary context. Feel free to use one
     or two fluent sentences to ensure the criterion is both complete and easy for a human to
      understand.
    For clarity with complex topics, feel free to use one or two fluent sentences.

3.  Output Format: Your response must be a valid JSON list of objects. Each object in the
     list must contain a single key, "criterion".
    The `criterion` string should be a natural, human-readable sentence that clearly states
     the verifiable fact and its scope.

4.  Escaping and Formatting Rules (Very Important): To ensure the output is valid JSON, you
     must follow these rules for the `criterion` string:
    Backslashes: All backslashes `\` MUST be escaped as `\\`. For example, `$\sigma$` must be
     written as `$\\sigma$`.
    Double Quotes: All double quotes `"` that are part of the text MUST be escaped as `\"`.
     For example, a "vanilla" model would be written as `a \"vanilla\" model`.

5.  Cross-reference the paper-level ENTITY MANIFEST below to complete criteria whose scope is
      defined ELSEWHERE in the paper. Some specifications describe a generic recipe/procedure
     /configuration WITHOUT naming which specific model(s)/dataset(s)/method(s) it applies to
     , because the paper states that scope in a DIFFERENT section (e.g. the recipe is in "
     Methods" but the model list is in "Experiment Setup" or an Appendix).
    A. PLAIN-LANGUAGE case (config/setup facts phrased in prose): when a specification is
     clearly model-dependent, dataset-dependent, or method-dependent (e.g. it fine-tunes "a
     pretrained model", evaluates "on the dataset", without itself naming which one(s)) and
     the manifest below lists multiple named entities for that axis, REWRITE the resulting
     criterion to explicitly enumerate the FULL scope from the manifest IN ONE sentence (e.g.
      "This recipe is applied separately for each of the following models: BART0-Large, BART0
     -XL, FLAN-T5-Large, FLAN-T5-XL, FLAN-T5-3B, using the dataset defined for each model.").
      Do NOT create one separate criterion per entity -- ONE consolidated, fully-enumerated
     criterion is correct and sufficient (this mirrors how a well-written reproduction plan
     would state it: naming the full scope once, not repeating the same paragraph per model).
    B. ABSTRACT-NOTATION case (equations/algorithms using symbols like $f_0$, $f$, $h$, an "
     encoder" function, $D$, $M$ that STAND FOR "the (base/pretrained) model", "the dataset",
      or "the method" as a role/placeholder, without naming a concrete instance): recognize
     this as EQUALLY entity-dependent as the plain-language case in A, even though the word "
     model"/"dataset" itself may not literally appear next to a name. Do NOT rewrite the
     formula/definition criterion itself (it must stay generic -- the formula does not change
      per entity, forcing a model name into it would be wrong). Instead, EMIT ONE EXTRA
     consolidated criterion (in addition to the normal atomic criteria from steps A/B above)
     stating that this equation/procedure is instantiated/run separately using each named
     entity from the manifest in the relevant role -- mirroring rule D's evaluation-script
     synthesis pattern. This keeps the formula-defining criteria clean while still making the
      concrete per-entity scope independently verifiable.
    For BOTH A and B: if the manifest's `combination_note` states exceptions (not a full
     cross product), reflect those exceptions precisely (e.g. "BART0 uses the P3 test split;
     FLAN-T5 uses MMLU instead") rather than assuming every entity combines with every other.
      If a specification already names its own specific scope (e.g. already says "for BART0-
     Large"), or is genuinely NOT entity-dependent (e.g. a general codebase/library fact, an
     environment/dependency fact), leave it exactly as steps 1-2 would otherwise produce --
     do NOT force-inject the manifest into criteria it doesn't apply to.

Paper-level entity manifest (JSON; use ONLY for the cross-referencing in step 5 above):
{manifest}
```

```
Here are three dummy examples of inputs and expected outputs.

Example 1: Handling Entities and Hyperparameters

Input
specification: "In our experiments, we use the AdamW optimizer with a learning rate of 1e-4
     and a weight decay of 0.01 on dataset Cora. we use the Adam with a learning rate of 2e-4
      and dropout of 0.2 on dataset Citeseer."

Your Output:
[
    {
      "criterion": "The AdamW optimizer is used to train the model for the dataset Cora."
    },
    {
      "criterion": "A learning rate of 0.0001 is applied when using the AdamW optimizer on
     the Cora dataset."
    },
    {
      "criterion": "A weight decay of 0.01 is used when using the AdamW optimizer on the Cora
      dataset."
    },
    {
      "criterion": "The Adam optimizer is used to train the model for the dataset Citeseer."
    },
    {
      "criterion": "A learning rate of 0.0002 is applied when using the Adam optimizer on the
      Citeseer dataset."
    },
    {
      "criterion": "A dropout of 0.2 is applied when using the Adam optimizer on the Citeseer
      dataset."
    }
]

Example 2: Handling Methods, Architecture, and Processes

Input
specification: "The actor loss for on-policy updates is the PPO clipped objective, defined as
      $L^{CLIP}(\theta) = \hat{\mathbb{E}}_t [ \min(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\
     theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t) ]$."

Your Output:
[
    {
      "criterion": "The actor loss is calculated using the PPO clipped objective: $L^{CLIP
     }(\\theta) = \\hat{\\mathbb{E}}_t [ \\min(r_t(\\theta) \\hat{A}_t, \\text{clip}(r_t(\\
     theta), 1 - \\epsilon, 1 + \\epsilon) \\hat{A}_t) ]$ for all on-policy updates."
    }
]
Example 3: Abstract Notation Standing For a Model (Entity-Manifest Cross-Reference, rule 5B)

Input
specification: "The ground truth forgetting binary indicator is $z_{ij} = 1$ if $f_0(x_i) \
     neq f_i(x_i)$, else 0, where $f_0$ is the base pretrained language model and $f_i$ is
     the model after being updated on example $\langle x_i, y_i \rangle$."
manifest: {"models": ["BART0-Large", "FLAN-T5-Large", "FLAN-T5-3B"], "datasets_tasks": [], "
     methods_baselines": [], "combination_note": ""}

Your Output (the formula-defining criterion stays generic; ONE extra criterion links it to
     the manifest per rule 5B):
[
    {
      "criterion": "The ground-truth forgetting binary indicator $z_{ij}$ is defined as 1 if
     $f_0(x_i) \\neq f_i(x_i)$ and 0 otherwise, where $f_0$ is the base pretrained model and
     $f_i$ is $f_0$ after being updated on example $\\langle x_i, y_i \\rangle$."
    },
    {
      "criterion": "This ground-truth-forgetting computation is implemented and run
     separately using each of BART0-Large, FLAN-T5-Large, and FLAN-T5-3B as the base
     pretrained model $f_0$."
    }
]

Example 4: Resource Acquisition and In-Training Logging (rule E)

Input
specification: "We fine-tune the pretrained RoBERTa-base checkpoint on the SST-2 dataset,
     tracking the training loss at every step until convergence."
```

```
Your Output (atomic facts from steps A/B, PLUS the extra rule-E criteria for the named
     resources and the in-training-observed quantity):
[
    {
      "criterion": "The pretrained RoBERTa-base checkpoint is used as the starting point for
     fine-tuning."
    },
    {
      "criterion": "The SST-2 dataset is used for the fine-tuning task."
    },
    {
      "criterion": "Training continues until the model converges."
    },
    {
      "criterion": "Code is implemented to download or load the pretrained RoBERTa-base
     checkpoint."
    },
    {
      "criterion": "Code is implemented to download or load the SST-2 dataset."
    },
    {
      "criterion": "The training loss is logged or tracked at every training step, not only
     at the end of training."
    }
]

Example 5: Multi-Condition Evaluation Protocol (rule F)

Input
specification: "We report the R^2 and MAE between in-distribution Top-1 accuracy and out-of-
     distribution Top-1 accuracy on ImageNet-V2 and ImageNet-Sketch, computed across all 75
     evaluated models."

Your Output (atomic per-condition facts from steps A/B/C, PLUS one evaluation-script
     criterion PER OOD test condition per rule F -- NOT one combined criterion):
[
    {
      "criterion": "$R^2$ is used as a metric between in-distribution and out-of-distribution
      Top-1 accuracy, computed across all 75 evaluated models."
    },
    {
      "criterion": "MAE is used as a metric between in-distribution and out-of-distribution
     Top-1 accuracy, computed across all 75 evaluated models."
    },
    {
      "criterion": "ImageNet-V2 is used as an out-of-distribution test set for the in-
     distribution vs. out-of-distribution accuracy comparison."
    },
    {
      "criterion": "ImageNet-Sketch is used as an out-of-distribution test set for the in-
     distribution vs. out-of-distribution accuracy comparison."
    },
    {
      "criterion": "An evaluation script computes and reports both the $R^2$ and MAE between
     in-distribution and ImageNet-V2 out-of-distribution Top-1 accuracy across all 75 models
     ."
    },
    {
      "criterion": "An evaluation script computes and reports both the $R^2$ and MAE between
     in-distribution and ImageNet-Sketch out-of-distribution Top-1 accuracy across all 75
     models."
    }
]
Please respond with ONLY the list.

specification:
{specification}
```

G.2.2 Stage 2: Entity-Axis Verification and Section Audit

Stage 2 re-mines the paper from the provisional checklist. Single-entity criteria are masked into templates (the entity is replaced by a placeholder, so criteria that share a fact but differ in the entity they name land in the same group); for each group the manifest supplies the candidate entities that still lack such a criterion, and BM25 retrieval supplies the paper excerpt that is checked for explicit support. The section audit then presents each implementation-relevant section with its most relevant existing items and asks for genuinely uncovered facts only.

Entity-axis verification prompt.

```
ENTITY_COVERAGE_VERIFICATION_PROMPT

You are auditing a research-paper reproduction CHECKLIST for INCOMPLETE coverage of a fact
     across all instances of one entity axis (e.g. across all models, or all datasets, or all
      baseline methods named in the paper).

You are given:
  - an EXAMPLE existing checklist criterion (verbatim) that states some fact/pattern for ONE
     entity on a given axis (model / dataset-task / method-baseline).
  - the axis type and the entities ALREADY covered by a criterion following this same fact/
     pattern.
  - a list of CANDIDATE entities (from the paper's full entity list for this axis) that do
     NOT yet have a criterion following this pattern.
  - an EXCERPT of the paper, retrieved because it is lexically related to this pattern.

For EACH candidate entity, decide whether the excerpt gives CLEAR, EXPLICIT evidence that the
      SAME underlying fact/pattern also applies to it. If so, WRITE a new, standalone,
     grammatically correct checklist criterion for that entity -- do NOT mechanically
     substitute the entity name into the example's exact wording (the example's exact
     phrasing may not grammatically fit another entity, especially if the example criterion
     itself enumerates multiple entities in one sentence, or the entity names share
     overlapping substrings/aliases). Instead, COMPOSE the new criterion the way a careful
     human writer would: same LEVEL of specificity and same fact/pattern as the example,
     phrased naturally and correctly for this specific entity, grounded ONLY in what the
     excerpt actually states.

STRICT RULES:
1. Only confirm+write an entity if the excerpt EXPLICITLY supports it for THAT entity -- do
     not assume that because a fact holds for one entity, it automatically holds for another
     without a real textual signal (papers frequently have per-entity exceptions -- this is
     exactly the kind of exception a careless assumption would miss).
2. If the excerpt doesn't mention the candidate entity in relation to this pattern at all, do
      NOT include it -- leave it out. Most candidates should be unconfirmed; that is expected
      and fine.
3. Be conservative: a false or fabricated addition is worse than missing a real gap.
4. The new criterion must be a complete, self-contained, well-formed sentence (or short
     sentences) -- never a fragment, never a mechanical find-and-replace artifact (e.g. never
      produce nonsense like a name duplicated inside another name, or a dangling half-
     rewritten list).
5. If the SAME excerpt happens to confirm the pattern for MULTIPLE candidate entities at once
     , still write ONE separate, complete criterion PER entity (never merge several entities
     back into one combined/listy sentence -- that recreates the original under-specification
      problem this process exists to fix).

Output ONLY a JSON object:
{"results": [{"entity": "<candidate entity name>", "new_criterion": "<the new, well-formed
     criterion text>", "evidence": "<short quote/paraphrase of what in the excerpt supports
     it>"}, ...]}
Only include entities you are confirming (omit unconfirmed ones entirely). If none are
     confirmed, output {"results": []}.
```

Section audit prompt.

```
SECTION_COVERAGE_AUDIT_PROMPT

You are auditing a research-paper reproduction CHECKLIST for COMPLETENESS against ONE SECTION
      of the paper it was built from.

You are given:
  - the raw text of ONE section of the paper.
  - a BATCH of existing checklist items (a small subset, retrieved because they are the most
     topically relevant existing items to this section -- NOT the whole checklist).

Carefully read EVERY item in the batch and cross-check it against the section text point by
     point -- with only ~10 items shown, you are expected to actually verify each one
     individually, not skim.

Find any VERIFIABLE, IMPLEMENTATION-RELEVANT fact stated in THIS SECTION that is NOT captured
      -- at ANY level of generality -- by ANY item in this batch. This includes (but is not
     limited to):
  - a specific formula/equation not stated by any shown item.
  - a specific algorithm/procedure step (or a whole missing step of an already-partially-
     covered multi-step procedure).
  - a specific hyperparameter, configuration value, or per-case value (e.g. a per-task/per-
     dataset/per-ablation number) not yet present.
  - a dataset/model/method identity, acquisition detail, or evaluation-protocol detail not
     yet present.
```

```
  - a standalone fact that has no clear existing analogue in this batch at all.

Do NOT flag something already captured by one of the shown items, even if worded differently
     or more generally -- only flag genuine gaps. Do NOT flag pure narrative/motivation
     sentences that don't correspond to an implementable or checkable fact (e.g. "this is
     challenging because...", "prior work has shown..."). Do NOT flag a claimed RESULT/
     OUTCOME value on its own UNLESS it is the kind of concrete number a reproduction's
     evaluation script would need to report (result values that only exist after running the
     experiment, like a bare accuracy number in a results table, are fine to include as an
     atomic fact the same way the rest of the checklist already does for such tables).

NOTE: because you only see a topically-relevant SUBSET of the full checklist (not everything)
     , a fact you flag here might turn out to already be covered by a DIFFERENT item
     elsewhere in the full checklist that isn't in this batch -- that's expected and handled
     separately after this step, so flag based ONLY on what you can see in this batch; do not
      try to guess what might exist elsewhere.

For each genuine gap found, WRITE ONE new, atomic, self-contained, well-formed checklist
     criterion capturing it -- following the SAME atomicity rules as the rest of the
     checklist:
  - a single self-contained equation/formula is ONE criterion (never split into its own sub-
     terms).
  - a multi-step procedure gets ONE criterion PER missing step (never a collapsed blob
     covering several steps).
  - distinct per-case values (different numbers for different tasks/datasets/ablations) are
     SEPARATE criteria, never merged.
  - ground every criterion STRICTLY in what this section's text actually states -- never
     fabricate or extrapolate.

Output ONLY a JSON object:
{"new_criteria": [{"criterion": "<new criterion text>", "evidence": "<short quote/paraphrase
     from the section supporting it>"}, ...]}
If nothing is missing (relative to this batch), output {"new_criteria": []}. Most batches
     should yield FEW or ZERO new criteria if the checklist is already thorough -- do not pad
      the output.
```

G.3 File-Scoped Code-Checklist Construction

G.3.1 File-Scoped Code-Checklist Prompts

Both rounds operate on one target file at a time and never read a candidate implementation or the reference repository. Round 1 drafts the checklist from the requirements the plan assigns to the file, consulting the paper through `consult_paper` only where the plan leaves a code-verifiable detail underspecified. Round 2 starts from that draft and emits only the items needed to close a gap it can actually name. Each item carries a provenance tag recording whether the plan alone, the plan plus retrieved paper text, or a paper-supported gap justifies it; round 2 items are deduplicated against round 1 items by exact text only.

Round 1: generator prompt.

```
FILE_CHECKLIST_GENERATION_PROMPT

You are extracting a code-verification checklist for ONE target file of a paper reproduction.

You are given the full reproduction PLAN (the source of truth for what each file must
     implement) and a target file path. You also have a tool, `consult_paper`, that retrieves
     passages from the original paper.

GROUND RULES
1. The PLAN defines the SCOPE: only produce requirements that the plan assigns to THIS file.
     The generated code is built from the plan, so do not invent requirements the plan never
     raises.
2. Use `consult_paper` ONLY when the plan is vague/approximate about a concrete, code-
     verifiable detail this file needs (an exact constant, a layer config, a hyperparameter
     value, a formula, a scope/condition). Use the paper to make a plan item PRECISE -- not
     to add unrelated requirements.
3. Make every item atomic, flat, and independently verifiable by reading THIS file's source
     code (a constant, signature, config value, API call, or import). Split multi-condition
     states; keep formulas whole; always include the concrete value AND its scope.

EXHAUSTIVENESS CHECKLIST (the most common ways a first pass under-covers a file -- check ALL
     of these against the plan text assigned to this file before finalizing):
- COMBINATORIAL GRID ENUMERATION: if the plan describes an operation run across a MULTI-
     DIMENSIONAL grid (e.g. {method} x {dataset} x {hyperparameter value} x {model size/
     variant}), do not emit one vague item per axis -- emit one item per COMBINATION actually
```

```
     named in the plan for this file. Never collapse a grid into a single generic item, and
     never truncate a list of combinations with "etc." or "and so on".
- NAMED-BASELINE / NAMED-COMPONENT PROVENANCE: for EACH externally-named baseline, method
     variant, or prerequisite/auxiliary model this file must implement or reproduce, emit a
     SEPARATE item (citing its source repo/reference/codebase if the plan gives one). Do not
     fold multiple named baselines or components into one item like "baselines are
     implemented". If the plan doesn't state what codebase a named baseline should be adapted
      from, PROACTIVELY consult_paper for it (a common paper convention: "all baselines are
     implemented based on X codebase") -- implementation-source fidelity for a named baseline
      this file must reproduce is a core requirement even when the plan's own text never
     raises it; this is a legitimate, expected use of `[paper-gap]`, not an edge case to use "
     sparingly".
- MINIMUM RUNTIME / ITERATION COUNT THRESHOLDS: if the plan states a MINIMUM duration/count
     requirement (e.g. "run for at least N iterations/steps/epochs"), emit this exact
     threshold as its own item for EACH method/config combination it applies to in this file
     -- don't state it once generically when it actually applies per-combination.
- EXACT DEFINITIONS & FORMULAS: for every metric, quantity, or algorithmic formula this file
     computes, emit an item asserting its EXACT definition/formula as stated in the plan (
     symbols, coefficients, conditions included) -- not just "computes metric X" or "applies
     the loss".
- NAMED-ENTITY LIST COMPLETENESS: if the plan gives an explicit list of named items (datasets
     , tasks, splits, classes, environments, prompts) this file must use or iterate over, the
      checklist item must name EVERY item in the list verbatim -- never summarize a list with
      "etc.", "among others", or a partial subset.
- EXACT VALUES FROM TABLES/APPENDICES: emit items for specific values that matter for this
     file (epochs, batch sizes, learning rates, key constants, thresholds), especially ones
     stated in a table or appendix -- don't drop a concrete value into a vague paraphrase.
- BOOKKEEPING / COST / LOGGING REQUIREMENTS: don't only check for the "main" method and
     metric requirements -- also scan for auxiliary requirements this file must satisfy (cost
     /resource tracking, logging intermediate values, saving artifacts for later analysis).
     These are easy to skip since they aren't the primary method or eval metric but are still
      explicit plan requirements.
- COMPLEX MULTI-CLAUSE PROTOCOLS: if the plan describes a procedure with several conditions/
     branches (e.g. "roll out until A, or B, or C happens"; a multi-step evaluation
     methodology), make sure your item(s) state ALL the clauses/conditions, not just the
     first or simplest one -- split into multiple items if one item can't cleanly hold every
     clause.

SELF-CHECK BEFORE FINALIZING (do this internally, in the same turn, before emitting the final
      block):
After drafting your checklist, RE-READ the plan text assigned to this file ONE MORE TIME
     specifically hunting for anything the exhaustiveness checklist above would catch that
     your draft still missed (an uncovered grid combination, a named baseline/list item you
     summarized away, an exact value you paraphrased, a bookkeeping requirement you skipped).
      Merge anything newly found into the final list before you output it. Do this silently
     -- do not narrate the re-check, just make sure the final list reflects it.

PROVENANCE TAG (prefix every item):
- `[plan]`      the item is fully specified by the plan.
- `[plan+paper]` the plan raised it; you sharpened the exact value/scope using the paper.
- `[paper-gap]`  the plan omits it but the paper clearly requires it for this file. Use
     SPARINGLY.

OUTPUT
For trivial files (empty __init__.py, .gitignore, LICENSE, pure re-export), output exactly:
<FILE_LEVEL_CHECKLIST>no need</FILE_LEVEL_CHECKLIST>

Otherwise:
<FILE_LEVEL_CHECKLIST>
### File: `<file_path>`
Purpose: <one sentence>

Checklist:
- [plan] <atomic requirement with concrete value + scope>
- [plan+paper] <atomic requirement ...>
...
</FILE_LEVEL_CHECKLIST>

When you have gathered enough detail, emit the final <FILE_LEVEL_CHECKLIST> block and stop.
     Do not call the tool after emitting it.
```

Round 2: completeness critic prompt.

```
FILE_CHECKLIST_CRITIQUE_PROMPT

You are a COMPLETENESS CRITIC reviewing an EXISTING code-verification checklist for ONE
     target file of a paper reproduction. A first pass already wrote the checklist below for
     this file; your job is to REVIEW and SUPPLEMENT it, NOT to regenerate it from scratch --
      regenerating would just duplicate the first pass's job.
```

```
You are given: the full reproduction PLAN (reference/verification material), the target file
     path, and the file's CURRENT checklist. You also have a `consult_paper` tool to pull
     passages from the original paper.

START FROM THE CHECKLIST, not the plan: go through the CURRENT checklist item by item and ask
     , for each one, "does this fully state what the plan (and paper) require for this file,
     with every combination/clause/value spelled out?" Cross-check against the plan text, and
      use consult_paper when the plan is vague about a concrete detail or when you suspect
     the paper states something the plan omits entirely (e.g. what codebase a named baseline
     should be adapted from). Emit ONLY the items needed to close a gap you actually found --
      do not re-emit or restate items that are already adequate, and do not invent
     requirements outside what the plan assigns to this file.

Hunt for these recurring GAP PATTERNS (the most common ways an existing checklist under-
     covers a file -- check ALL of them against the current checklist):
- COMBINATORIAL GRID ENUMERATION: if this file's work spans a MULTI-DIMENSIONAL grid (e.g. {
     method} x {dataset} x {hyperparameter value} x {model size/variant}), check which named
     combinations already have an item and which don't -- emit one item per MISSING
     combination. This is the single most common under-coverage pattern. Never truncate a
     list of combinations with "etc.".
- NAMED-BASELINE / NAMED-COMPONENT PROVENANCE: for EACH externally-named baseline, method
     variant, or prerequisite/auxiliary model this file implements, check there's a separate
     item citing its source repo/reference/codebase (if the plan gives one, or the paper does
      via consult_paper -- a common paper convention is "all baselines are implemented based
     on X codebase"). Implementation-source fidelity for a named baseline is a core
     requirement even if the plan text never explicitly raises it.
- MINIMUM RUNTIME / ITERATION COUNT THRESHOLDS: if a minimum duration/count requirement
     applies (e.g. "run for at least N iterations/steps/epochs"), check it's asserted for
     EACH method/config combination this file handles, not just stated once generically.
- EXACT DEFINITIONS & FORMULAS: for each metric/quantity/algorithmic formula this file
     computes, check the checklist asserts its EXACT definition (symbols, coefficients,
     conditions) -- not a vague paraphrase like "computes metric X".
- NAMED-ENTITY LIST COMPLETENESS: if this file uses/iterates an explicit list of named items
     (datasets, tasks, splits, classes, environments, prompts), check EVERY item in that list
      is named verbatim in the checklist, not a summarized/partial subset.
- EXACT VALUES FROM TABLES/APPENDICES: check specific values (epochs, batch sizes, learning
     rates, key constants, thresholds) that matter for this file are captured verbatim, not
     paraphrased away.
- BOOKKEEPING / COST / LOGGING REQUIREMENTS: check auxiliary requirements (cost/resource
     tracking, logging intermediate values, saving artifacts) this file must satisfy aren't
     skipped just because they aren't the primary method/metric.
- COMPLEX MULTI-CLAUSE PROTOCOLS: if this file implements a procedure with several conditions
     /branches (e.g. "roll out until A, or B, or C happens"; a multi-step evaluation
     methodology), check the checklist states ALL the clauses, not just the first/simplest
     one.

THIS IS YOUR ONLY REVIEW PASS OVER THIS FILE'S CHECKLIST -- there will be no follow-up round,
      so be exhaustive in this single turn. Work in two internal steps (do not narrate them,
     just do the work silently):
  STEP A -- DRAFT: read the plan + current checklist once, draft supplementary items using
     the GAP PATTERNS above.
  STEP B -- RE-SCAN: re-read the CURRENT checklist ONE MORE TIME, going through the GAP
     PATTERNS list one by one, assuming your draft missed something. Merge new findings
     before finalizing.

OUTPUT
If nothing needs to be added, output exactly:
<FILE_LEVEL_CHECKLIST>none needed</FILE_LEVEL_CHECKLIST>

Otherwise, output ONLY the new supplementary items (do not repeat existing ones):
<FILE_LEVEL_CHECKLIST>
- <new supplementary item, self-contained and binary-gradable>
- <new supplementary item>
...
</FILE_LEVEL_CHECKLIST>
```

G.4 GRM SCORING

G.4.1 GRM PROMPTS

Plan pipeline.

For a plan-pipeline rollout the GRM receives the final candidate plan and one checklist criterion, and returns a binary verdict. Partial credit is explicitly disallowed so that the plan reward equals the fraction of criteria the plan fully specifies.

```
PLAN_CRITERION_JUDGMENT_PROMPT

You judge whether a paper-reproduction PLAN specifies a single REQUIREMENT precisely enough
     to implement without re-reading the paper.

You are given:
  - the FULL candidate plan document, and
  - one REQUIREMENT (something a good plan must state, often with an exact value/formula/
     scope) to check against it.

Decide one verdict:
  - "satisfied"     : the plan explicitly states the requirement WITH the exact value /
     formula / scope called for. A reader would not need to guess or re-read the paper.
  - "not_satisfied" : the plan does NOT state the requirement precisely enough -- this covers
      not addressing it at all, only vaguely gesturing at the general topic without the
     concrete value/formula/scope, or contradicting it. There is no partial credit.

STRICT RULES (do not relax):
1. Do NOT give "satisfied" credit for a plan that merely uses the same keywords as the
     requirement without stating the actual value/formula/scope. Naming a concept is not the
     same as specifying it.
2. A vague paraphrase of the paper's own wording ("the loss described in the paper") without
     restating the formula/value is "not_satisfied".
3. Judge ONLY from the supplied full candidate plan. If the specifying detail is absent, do
     not assume it is implied or available elsewhere.
4. Longer or more verbose plan text is NOT evidence of satisfaction -- judge strictly on
     whether the SPECIFIC requirement is met.

Return ONLY a JSON object:
{"verdict": "satisfied"|"not_satisfied", "reason": "<one short sentence citing what the plan
     does/doesn't state>"}
```

File-scoped code writing.

For file-scoped code-writing rollouts, training uses `FILE_CRITERION_JUDGMENT_PROMPT` below. The GRM judges one criterion against the complete target file, using source-code logic rather than identifier overlap.

```
FILE_CRITERION_JUDGMENT_PROMPT

You are a code verification expert bridging the gap between theoretical research papers and
     practical software engineering. Your task is to analyze source code and determine if it
     correctly and executably implements a specific checklist requirement.

You will receive:
1. Criterion: a single atomic requirement from a file-level checklist.
2. Source Code: the complete implementation file.

Determine if the code correctly implements this criterion.

Verification guidelines:
1. Analyze logic rather than identifier names and check that the code belongs to the relevant
      pipeline stage.
2. If a specific value is required, accept it when set directly or exposed as a configurable
     option; reject a conflicting hard-coded value with no way to change it.
3. Accept standard engineering or theoretical equivalences, but reject fundamental
     algorithmic conflicts, placeholders, unfinished code, or broken execution paths.

Return true only when the source code correctly implements the criterion. Return false for
     absent, incorrect, incomplete, contradictory, placeholder, or unverifiable
     implementations. Do not rely solely on comments.

Return ONLY a JSON object:
{"verdict": true|false, "evidence": "Quote exact code or explain equivalence."}
```

G.5 CURATION VERIFICATION

G.5.1 CURATION VERIFIER PROMPTS

Curation verifies each atomic reproduction criterion against the paired repository in a retrieve-then-read manner. The first prompt localizes candidate files from the repository's structural views (directory tree, file/class/function summaries, import dependencies); the second reads the retrieved code and decides whether the criterion is implemented. The verdicts feed the importance-weighted alignment score of Eq. 1.

Retrieval prompt (criterion to candidate files).

```
CURATION_FILE_RETRIEVAL_PROMPT

You are a code analysis expert. Your task is to identify which source code files are most
     likely to contain the implementation described in a given criterion.

Based on the criterion and the project information provided, identify the TOP 3 most relevant
      Python files (.py) where this criterion is most likely implemented.

Consider:
1. File names and their semantic meaning from the directory tree
2. Function/class summaries and their descriptions
3. File dependencies and relationships
4. The specific technical details mentioned in the criterion

The project information includes:
- Project Directory Tree: A clean directory structure showing file organization
- Code Summaries: Detailed functional summaries for each file, class, and function
- File Dependencies: Inter-file dependency relationships within the project

Output Format
Respond with ONLY a JSON object containing a "matched_files" key with an array of exactly 3
     file paths (relative to project root), ordered by relevance (most relevant first).

Example output format:
{
  "matched_files": ["classifiers/AoA_model.py", "classifiers/classifiers.py", "model_training
     /MLP_train.py"]
}

Important:
- Return ONLY the JSON object, no additional text or explanation
- Include exactly 3 file paths in the "matched_files" array
- Use relative paths from the project root
- Only include .py files that exist in the project structure
```

Verification prompt (criterion against retrieved code).

```
CURATION_CODE_VERIFICATION_PROMPT

You are a code verification expert. Your task is to carefully analyze source code files and
     determine whether they correctly implement a specific criterion extracted from a
     research paper.

Your Task

You will be given:
1. A criterion - a specific implementation requirement from a research paper
2. Source code files - the actual implementation code to verify

You must determine whether the provided source code correctly implements the criterion.

Verification Guidelines

What to Check:

1. Exact Match: Does the code implement the exact value, parameter, or configuration
     mentioned in the criterion?
   - For hyperparameters: check learning rates, batch sizes, epochs, etc.
   - For architecture: check layer sizes, neuron counts, model structures, etc.
   - For data processing: check split ratios, preprocessing steps, etc.

2. Code Evidence: Is there clear, unambiguous code that demonstrates the implementation?
   - Look for variable assignments, function arguments, configuration dictionaries
   - Check comments or docstrings that confirm the implementation
```

```
   - Verify the values are actually used in the execution flow

3. Context Matching: Does the implementation context match the criterion's scope?
   - If the criterion specifies "for training", verify it's in training code
   - If it mentions a specific model (e.g., "for BERT"), verify it's in that model's code

What NOT to Do:

- Do NOT mark as True if the value is different from what the criterion specifies
- Do NOT mark as True if you only find similar but not exact implementations
- Do NOT make assumptions - if the code is unclear or ambiguous, mark as False
- Do NOT mark as True based on variable names alone without seeing the actual values

Edge Cases:

- Multiple implementations: If there are multiple values for the same parameter (e.g., in
    different functions), verify that at least one matches the criterion exactly
- Default values: If the criterion mentions a "default" configuration, verify it's actually
    the default in the code
- Comments vs. Code: Prioritize actual code over comments. Code is the ground truth.

Output Format

You must respond with ONLY a JSON object in this exact format:

{
  "verdict": true,
  "evidence": "Exact quote or description of the code that proves the verdict"
}

Field Descriptions:

- verdict:
  - `true` if the code clearly and correctly implements the criterion
  - `false` if the code does not implement it, implements it incorrectly, or if it's unclear

- evidence:
  - Quote the exact line(s) of code that support your verdict
  - Include variable names, values, and context
  - If verdict is false, explain what's missing or different

Examples

Example 1: Clear Match (TRUE)

Criterion: "A learning rate of 5e-5 is applied during fine-tuning."

Code:
```python
optimizer = Adam(lr=5e-5)
```

Response:
{
  "verdict": true,
  "evidence": "optimizer = Adam(lr=5e-5)"
}

Example 2: Value Mismatch (FALSE)

Criterion: "The batch size is set to 32."

Code:
```python
batch_size = 16
dataloader = DataLoader(dataset, batch_size=batch_size)
```

Response:
{
  "verdict": false,
  "evidence": "batch_size = 16"
}

Example 3: Ambiguous/Missing (FALSE)

Criterion: "The model uses dropout with rate 0.1."

Code:
```python
class Model(nn.Module):
```
```

```
    def __init__(self):
        self.dropout = nn.Dropout()
```

Response:
```
{
  "verdict": false,
  "evidence": "self.dropout = nn.Dropout() - no explicit rate specified"
}
```

Important Reminders

- Be strict: Only mark as `true` if you have clear, unambiguous evidence
- Quote exact code: Your evidence should be directly verifiable
- Consider scope: Make sure the implementation is in the right context
- Focus on facts: Base your verdict on what's actually in the code, not what you think should be there

Now, carefully analyze the criterion and code provided, and give your verdict.